Optimization Methods and Applications on problem solving with MATLAB in the presence of Randomness

Daniel James Taylor

Supervisor: Dr. Dimitra Antonopoulou

Mathematics Department

Thesis submitted to the

University of Chester

Abstract

A review of iterative methods used to find optimal solutions to large sparse linear systems including methods based on line search descent algorithms and Krylov subspace methods. We also detail how to use the MATLAB optimization toolbox to solve a variety of optimization problems including linear and non-linear problems in Chapter 2. A review of the classical Travelling Salesman Problem (TSP) is provided in Chapter 3 with examples of solved problems. In Chapter 4 we used a MATLAB program to investigate the effect that randomness has on a system of ODE's namely the equation of a harmonic pendulum, we demonstrate these effects with a number of plots in the phase-plane and with respect to the time t.

October 7, 2017

Acknowledgements

I would first like to thank my supervisor throughout the process of writing this thesis, Dr. Dimitra Antonopoulou for her suggestions and guidance in carrying out this piece of work. I would also like to acknowledge all of the staff in the Mathematics department at the University of Chester and would like to place on record my thanks to the University of Chester for allowing me to carry out this work.

Table of Contents

Introduction

The first Chapter in this thesis aims to provide a review of various algorithms relating to line search decent methods and Krylov subspace methods. In Chapter 1, we provide a detailed review of such methods including the classical steepest descent and the very useful conjugate gradient method, which are optimization methods for solving linear or non-linear system, where the initial problem is equivalently transformed to the problem of optimizing a linear or non-linear functional. Note that we will only present these methods in their simplest form as there are many more modified versions of such methods that aim to improve convergence for certain problems that are particularly large and difficult.

Many of these methods are used for solving large and sparse linear and non-linear systems. Much of the work in this chapter is inspired by that of Saad [11] and Snyman [12].

In Chapter 2 the aim is to provide an insight in to optimization techniques and how they can be implemented in MATLAB, specifically much of the focus in Chapter 2 is solving optimization problems in MATLAB's optimization toolbox [9]. An inbuilt GUI in MATLAB used specifically for solving all types of optimization problems; it deploys various algorithms and solvers. We cover some of these in more detail in Chapter 2, including examples of how to use the optimization toolbox to solve all types of optimization problems. [8] proved particularly useful when formulating and solving such problems.

The Travelling Salesman Problem (TSP) is one of the most classical linear optimization problems in mathematics [7]. In Chapter 3, we describe the problem and how it can be formulated to look like an optimization problem (solvable with the Simplex method) as well as some of the most well known methods for solving the TSP. We also provide some interesting solutions to different sized optimization problems. The aim of Chapter 3 is to provide an insight in to the work that goes in to solving NP hard problems.

Monte Carlo methods is a very important family of numerical methods [14], mainly used when we want to insert some element of randomness in to a process. We might see this in physical or biological phenomena and it is important to be able to represent randomness mathematically. In Chapter 4 we apply a Monte Carlo approach to two examples, the second of which is a harmonic pendulum ODE system. We then study the effects that introducing randomness has on such a system. In this chapter we present many experiments formulated in MATLAB to show how changing various parameters and increasing/decreasing the amount of randomness in a system effects the qualitative behaviour of the solution.

Chapter 1

Line search descent methods & Krylov subspace methods

Used to solve unconstrained minimization problems, line search descent methods are iterative methods designed to find the local minimum of a real function f of one or more real variables.

We start with some initial guess and then produce a sequence of iterative values; the stopping criterion for experimental convergence of the method to the local minimum is when there is no or very little change (up to a tolerance) between one iterate and the next.

The algorithm given below adapted form that found in chapter 2 of [12] describes the general iterative process that all line search methods follow.

1. Start with some initial guess, lets say $\boldsymbol{x}^0$ and positive tolerances ϵ_1, ϵ_2, and ϵ_3.

2. Choose the direction of descent $\boldsymbol{u}^i$ such that the following condition holds

$$\frac{df(\boldsymbol{x}^i)}{d\lambda}\bigg|_{u^{i+1}} = \boldsymbol{\nabla}^T f(\boldsymbol{x}^i)\boldsymbol{u}^{i+1} < 0.$$

   This ensures that we get descent at $\boldsymbol{u}^i$ in the direction of $\boldsymbol{u}^{i+1}$. Here $\boldsymbol{\nabla}$ denotes the gradient.

3. Perform a one-dimensional line search in the direction $\boldsymbol{u}^i$ i.e. find real λ such that

$$\min_{\lambda} F(\lambda) = \min_{\lambda} f(\boldsymbol{x}^{i+1} + \lambda \boldsymbol{u}^i).$$

4. Set $\boldsymbol{x}^i = \boldsymbol{x}^i + \lambda_i \boldsymbol{u}^i$ and then test for convergence using the following conditions

$$\text{if } ||\boldsymbol{x}^i - \boldsymbol{x}^{i+1}|| < \epsilon_1 \text{ or } ||\boldsymbol{\nabla} f(\boldsymbol{x}^i)|| < \epsilon_2 \text{ or } |f(\boldsymbol{x}^i) - f(\boldsymbol{x}^{i-1})| < \epsilon_3$$

   then STOP. We have $\boldsymbol{x}^* = \boldsymbol{x}^i$, else continue to step 5.

5. Set $i = i + 1$ and return to step 2. Repeat the procedure until we get convergence, i.e. until one or more of the above conditions in step 4 holds.

Now that we have outlined the general structure of a line search descent method, we present in more detail two of the most important methods of this kind: the steepest descent method (also known as the gradient descent method), and the conjugate gradient method.

1.1 Steepest descent method

One of the most famous line search descent methods, is the steepest descent method, and was first proposed by Cauchy (1847).

The optimization problem considered is the unconstrained minimization problem defined by

$$\text{specify } \boldsymbol{x} \in R^n : \quad f(\boldsymbol{x}) = \min_{x \in R^n} f(\boldsymbol{x}),$$

where $f(\boldsymbol{x})$ is a continuously differentiable function in the set R^n; also, we are interested in approximating the minimum.

Let $\boldsymbol{x}^i$ be the current iteration point where the initial guess is denoted by $\boldsymbol{x}^0$ and let

$$\boldsymbol{u}^i = \boldsymbol{u}(\boldsymbol{x}^i) = \boldsymbol{\nabla} f(\boldsymbol{x}^i),$$

be the gradient vector at $\boldsymbol{x}^i$. Then the next iterate is defined by

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i - \lambda_i \boldsymbol{u}^i,$$

where $\lambda^i > 0$ satisfies the following relation

$$f(\boldsymbol{x}^i - \lambda_i \boldsymbol{u}^i) = \min_{\lambda} f(\boldsymbol{x}^i - \lambda \boldsymbol{u}^i).$$

Example 1.11.

Let us now present a simple example, based on the one found in [12] of the steepest descent method.

We aim to minimize in R^2 the function

$$f(x, y) = 4x^2 - 4xy + 2y^2 \tag{1.1}$$

starting with an initial guess of $\boldsymbol{x}^0 = (2, 3)$.

Note that it can be easily seen that the minimum here is taken at $(0, 0)$ and equals 0. We will attempt to verify this using the steepest descent method (in order to observe the convergence of this method to the minimum).

Computing the steepest descent direction we find that, by taking the derivative of f with respect to x and then y we have

$$\nabla f(x, y) = (8x - 4y, 4y - 4x)$$

which when evaluated at $\boldsymbol{x}^0$ yields

$$\nabla f(\boldsymbol{x}^0) = \nabla f(2, 3) = (4, 4).$$

Now letting

$$g(\lambda) = f((2, 3 - \lambda(4, 4)) = f(2 - 4\lambda, 3 - 4\lambda),$$

we obtain

$$\begin{aligned} g'(\lambda) &= -\nabla(2 - 4\lambda, 3 - 4\lambda) \cdot (4, 4) \\ &= -(16 - 32\lambda - 12 + 16\lambda, 12 - 16\lambda - 8 + 16\lambda) \cdot (4, 4) \\ &= -(-16\lambda + 4, 4) \cdot (4, 4) \\ &= 64\lambda - 32, \end{aligned}$$

which has a global minimum at the point where $\lambda = \frac{1}{2}$.

We then continue the iterative process by setting

$$\boldsymbol{x}^1 = \boldsymbol{x}^0 - \frac{1}{2}\nabla f(\boldsymbol{x}^0) = (2,3) - \frac{1}{2}(4,4) = (0,1).$$

Then performing the next step of the algorithm we get

$$\nabla f(\boldsymbol{x}^1) = \nabla f(0,1) = (-4,4)$$

and thus,

$$g(\lambda) = f((0,1) - \lambda(-4,4)) = f(4\lambda, 1-4\lambda)$$

meaning we obtain the following

$$\begin{aligned} g'(\lambda) &= -(32\lambda - 4 + 16\lambda, 4 - 16\lambda - 16\lambda) \cdot (-4,4) \\ &= -(48\lambda - 4, -32\lambda + 4) \cdot (-4,4) \\ &= 320\lambda - 32 \end{aligned}$$

which has a global minimum when $\lambda = \frac{1}{10}$. So, the next step is given by

$$\boldsymbol{x}^2 = \boldsymbol{x}_1 - \frac{1}{10}\nabla f(\boldsymbol{x}_1) = (0,1) - \frac{1}{10}(-4,4) = \left(\frac{2}{5}, \frac{3}{5}\right).$$

By completing the next step of the algorithm we will be lead to $\boldsymbol{x}^3 = \left(0, \frac{1}{5}\right)$. Clearly we can see that we have a sequence that is converging towards $(0,0)$ so we can stop here and we have verified that indeed the minimum of the function f is given by $\boldsymbol{x}^* = (0,0)$. Next we will give some important results related to the steepest descent methods. The theorems and proofs here are based on those found in [1].

Definition 1.12.

A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is called L-Lipschitz if and only if

$$||\nabla f(x) - \nabla f(y)||_2 \leqslant L||x-y||_2, \qquad \forall x, y \in \mathbb{R}^n.$$

We let C_L denote the class of L-Lipschitz functions

Lemma 1.13.

If $f \in C_L$, then $|f(y) - f(x) - \langle \nabla f(x), y - x\rangle| \leqslant \frac{L}{2}||y-x||^2$. We will use this result to prove the following theorem related to the convergence of the steepest descent method with fixed step size.

Theorem 1.14.

If $f \in C_L$ and $f^* = -\min\limits_x f(x) > -\infty$, then the steepest descent method with fixed step size satisfying $\lambda < \frac{2}{L}$ will converge to a stationary point.

Proof:

Let $x^{i+1} = x^i - \lambda \nabla f(x^i)$. Then by the fact that $f \in C_L$ and using Lemma 1.13, we can derive the following

$$\begin{aligned} f(x^{i+1}) &\leqslant f(x^i) + \left\langle \nabla f(x^i), x^{i+1} - x^i \right\rangle + \frac{L}{2}||x^{i+1} - x^i||^2 \\ &= f(x^i) - \lambda ||\nabla f(x^i)||^2 + \frac{\lambda^2 L}{2}||\nabla f(x^i)||^2 \\ &= f(x^i) - \lambda \left(1 - \tfrac{\lambda}{2}L\right) ||\nabla f(x^i)||^2 \end{aligned}$$

rearranging then yields the following

$$||\nabla f(x^i)||^2 \leqslant \frac{1}{\lambda \left(1 - \frac{\lambda L}{2}\right)} \left(f(x^i) - f(x^{i+1})\right).$$

Hence, by summation, we arrive at

$$\begin{aligned} \sum_{k=1}^{N} ||\nabla f(x^{(k)})||^2 &\leqslant \frac{1}{\lambda \left(1 - \frac{\lambda L}{2}\right)} \left(f(x^{(0)}) - f(x^{(N)})\right) \\ &\leqslant \frac{1}{\lambda \left(1 - \frac{\lambda L}{2}\right)} \left(f(x^{(0)}) - f^*\right), \end{aligned}$$

which yields that

$$\lim_{k \to \infty} \nabla f(x^{(k)}) = 0$$

implying convergence; this completes the proof.

Theorem 1.15.

Let $\{x^i\}$ be the sequence generated by the steepest descent method algorithm. Then for all i, $x^{i+1} - x^i$ is orthogonal to $x^{i+2} - x^{i+1}$.

Proof

By the steepest descent algorithm, we have the following relations for x^{i+1} and x^{i+2}

$$x^{i+1} = x^i - \lambda_i \nabla f(x^i), \tag{1.2}$$

$$x^{i+2} = x^{i+1} - \lambda_{i+1} \nabla f(x^{i+1}). \tag{1.3}$$

So, we can write

$$\langle x^{i+1} - x^i, x^{i+2} - x^{i+1} \rangle = \lambda_i \lambda_i + 1 \langle \nabla f(x^i), \nabla f(x^{i+1}) \rangle \tag{1.4}$$

Now, recalling that λ_k minimizes the function g_i where

$$g_i(\lambda) = f(x^i - \lambda \nabla f(x^i)).$$

Observe that the following holds

$$\frac{dg(\lambda_i)}{d\lambda} = 0.$$

By chain rule we have that

$$\frac{dg_i}{d\lambda}(\lambda_k) = \langle -\nabla f(x^i), \nabla f(x^i - \lambda_i \nabla f(x^i)) \rangle \tag{1.5}$$

$$= \langle -\nabla f(x^i), \nabla f(x^{i+1}) \rangle \tag{1.6}$$

Now, by combining the above relationswe derive that

$$\langle x^{i+1} - x^i, x^{i+2} - x^{i+1} \rangle = 0,$$

which concludes the proof.

Definition 1.16. (Convex function)

A function $f(x)$ is said to be convex on some interval $[a, b]$ if for any two points x_1 and x_2 in $[a, b]$ the following holds

$$f[\lambda x_1 + (1 - \lambda)x_2] \leqslant \lambda f(x_1) + (1 - \lambda) f(x_2), \tag{1.7}$$

for any $0 < \lambda < 1$.

For convex functions that satisfy condition (1.7) in definition 1.16, the steepest descent method works well, namely all local minima are also global minima so the method will converge to the desired global solution. However the steepest descent method has some limitations, particularly with functions that are non convex. In this case it can often be found that we get a zig-zagging effect and the method will be extremely slow to converge. For this reason the steepest descent method is often seen as inferior to other line search descent methods for certain types of functions. We will next present at a more popular choice of method, the so-called conjugate gradient method.

1.2 Krylov Subspaces

In this section we will study methods related to Krylov subspaces including Arnoldi's method, Lanczos method, GMRES (Generalized Minimum Residual) method and the conjugate gradient method. All of these methods are used to find eigenvalues of large sparse matrices, and solving large linear systems where we want to avoid using matrix operations.

Definition 1.21. (Krylov Subspaces)

Consider an $n \times n$ matrix A and a vector b with dimension n. Then the order-r Krylov subspace is the linear subspace spanned by the images of b under the first r powers of A starting form $A^0 = I$, that is

$$K_r(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{r-1}b\}.$$

The sequence K_r converges to the eigenvector corresponding to the largest eigenvalue of A. However, this procedure is unstable, hence why we need to make use of Arnoldi's method and other related methods.

Definition 1.22. (Hessenberg Matrices)

Let A be an $n \times n$ matrix with zero entries below the lower sub-diagonal i.e.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2(n-1)} & a_{2n} \\ 0 & a_{32} & a_{33} & \dots & a_{3(n-1)} & a_{3n} \\ 0 & 0 & a_{43} & \dots & a_{4(n-1)} & a_{4n} \\ 0 & 0 & 0 & \ddots & a_{5(n-1)} & a_{5n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & 0 & 0 & 0 & a_{n(n-1)} & a_{nn} \end{pmatrix}$$

then A is called an upper Hessenberg matrix. Matrices with zero entries above the upper sub-diagonal are called lower Hessenberg.

1.3 Arnoldi's iterative method

Consider an $m \times m$ matrix A. Our first job is to build the Krylov matrix using power iteration. We take a random b and multiply it by $A, A^2, \dots A^n$. This forms the Krylov subspace we defined earlier. Now let us take this as a matrix where each column represents the vectors $b, Ab, \dots A^{n-1}b$. This is called the Krylov matrix

$$K_n = [b \quad Ab \quad A^2b \quad \dots \quad A^{n-1}b].$$

We then extract an orthogonal basis from the matrix K_n, usually using the Gram-Schmidt orthogonalization procedure. The vectors in this basis approximate the eigenvectors corresponding to the n largest eigenvalues of the matrix A.

Let us present now the algorithm for Arnoldi's method.

Arnoldi's method algorithm

Here we present the algorithm for the method, motivated by this presented by Saad in [11].

1. Choose some vector v_1 with norm 1.
2. For $j = 1, 2, \dots, m$, do the following:
3. Compute $h_{ij} = (Av_j, v_i)$ for $i = 1, 2, \dots, j$.
4. Compute $w_j = Av_j - \sum_{i=1}^{j} h_{ij} v_i$.
5. If $h_{j+1,j} = ||w_j||_2 = 0$ then stop.
6. Set $v_{j+1} = \dfrac{w_j}{h_{j+1,j}}$.
7. End.

   We have constructed thus, the basis which approximates the eigenvectors of A.

Notice that this algorithm is quite simple. We essentially just use the computed v_j at each step, multiply it by A and then orthonormalize the resulting w_j against all other v_i's creating the orthonormal basis of the Krylov matrix K_n. Next we will prove that indeed the algorithm does produce an orthonormal basis of K_n provided that the algorithm reaches the m^{th} step without stopping.

The following proof is again based on that found in [11].

Theorem 1.31.

Assume that Arnoldi's algorithm (given above) reaches the m^{th} step, i.e. $h_{j+1,j} \neq 0$ for all $j = 1, 2, \ldots, m$. Then the initially chosen vector v_1 and subsequent generated vectors $v_2, v_3, \ldots, v_m$ form an orthonormal basis of the Krylov subspace defined by

$$K_m = span\{v_1, Av_1, \ldots A^{m-1}v_1\}.$$

Proof:

The vectors $v_1, v_2, \ldots, v_m$ are clearly orthonormal, since we constructed them using the Gram-Schmidt process and normalized accordingly. The fact that these vectors span the Krylov subspace K_m follows from the fact that each of the vectors $v_1, v_2, \ldots, v_M$ is of the form

$$q_{j-1}(A)v_1$$

where q_{j-1} is a polynomial of degree $j-1$.

We show this using induction.

First note that the result clearly holds true for $j = 1$ since $v_1 = q_0(A)v_1$ with $q_0 = 1$.

Now, in the next step of induction assume, that the result also holds true for all $x \leqslant j$, where $x = 1, 2, \ldots, j$. Now consider the following,

$$\begin{aligned} h_{j+1}v_{j+1} &= Av_j - \sum_{i=1}^{j} h_{ij}v_i \\ &= Aq_{j-1}(A)v_1 - \sum_{i=1}^{j} h_{ij}q_{i-1}(A)v_1, \end{aligned}$$

which shows that indeed, V_{j+1} can be expressed as $q_j(A)v_1$ where q_j is a polynomial of degree j, which completes the proof by induction.

1.4 Lanczos method

A special case of Arnoldi's method, is the Lanczos method, being essentially a simplified version of the Arnoldi algorithm, that we can apply when A is an $n \times n$ symmetric matrix.

We much prefer this method when we have a symmetric matrix because the process takes advantage of the symmetry in the problem which cuts down the computational time considerably.

Definition 1.41. (Symmetric Matrices)

Let $A \in \mathbb{R}^{n\times n}$ be an $n \times n$ matrix, then A is said to be symmetric if $A = A^T$ where A^T denotes the transpose of A [6].

Example 1.42.

The following matrices are said to be symmetric since they are equal to their own transpose

$$A = \begin{pmatrix} 1 & -1 & 5 \\ -1 & 2 & -1 \\ 5 & -1 & 3 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 2 & 6 \\ 2 & 1 & 2 \\ 6 & 2 & 1 \end{pmatrix}.$$

Definition 1.43. (Tridiagonal Matrices)

Let $A \in \mathbb{R}^{n\times n}$ be an $n \times n$ matrix, then A is said to be tridiagonal if the only non-zero entries are the main diagonal, the diagonal above the main diagonal and the diagonal below the main diagonal [6].

Example 1.44

The following matrix is said to be tridiagonal since it has non-zero entries in the main diagonal and the diagonals directly above and below the main diagonal.

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 2 & 2 & -1 & 0 \\ 0 & -3 & 3 & 5 \\ 0 & 0 & -1 & 4 \end{pmatrix}.$$

Notice that an interesting property of tridiagonal matrices is that they are both upper and lower Hessenberg as can be seen in Definitions 1.18 and 1.22.

Before we present the algorithm for the Lanczos method we will first provide the following theorem and proof which helps us formulate the Lanczos algorithm adapted from that found in [11].

Theorem 1.45.

First assume that we have applied the Arnoldi's algorithm given earlier in this chapter to a real symmetric matrix A. Then the coefficients h_{ij} generated by applying the algorithm have the following properties

$$\begin{aligned} &h_{ij} = 0, \quad \text{for} \quad 1 \leqslant i < j-1, \\ &h_{j,j+1} = h_{j+1,j} \quad \text{for} \quad j = 1, 2, \ldots, m. \end{aligned} \tag{1.8}$$

The conditions (1.8) imply that H_m is both symmetric and tridiagonal.

Proof:

Given that $H_m = V_m^T A V_m$ we can say immediately that it is in fact symmetric. By construction we also know that H_m is a Hessenberg matrix (see Definition 1.18.). For these reasons we know that H_m must also be tridiagonal. This concludes the proof.

Lanczos algorithm

Here we present the Lanczos algorithm; notice the similarity to the Arnoldi algorithm we presented earlier.

For simplicity let us denote $\alpha_j = h_{jj}$ and $\beta_j = h_{j-1,j}$.

1. Choose an initial vector v_1 with norm 1. Also set $v_0 = 0$ and $\beta_1 = 0$.
2. For $j = 1, 2, \ldots, m$ do the following:
3. $w_j = Av_j - \beta_j v_{j-1}$.
4. $\alpha_j = (w_j, v_j)$.
5. $w_j = w_j = \alpha_j v_j$.
6. If $\beta_{j+1} = ||w_j||_2 = 0$ then stop.
7. $v_{j+1} = \dfrac{w_j}{\beta_{j+1}}$.
8. End, the algorithm is complete.

It can be seen that the main difference between Lanczos algorithm and Arnoldi's algorithm is that when using Lanczos algorithm we obtain H_m that is tridiagonal, hence we only need to store 3 vectors making the algorithm much more efficient. For this reason if we have an A that is symmetric then we always prefer the Lanczos procedure over that of Arnoldi.

1.5 GMRES (Generalized Minimum Residual) Method

The GMRES method is another iterative procedure. The GMRES method is particularly useful for solving systems of linear equations that are not symmetric. We make use of Arnoldi's iterative method in order to find an form the orthonormal basis before some further computations to solve our system.

Once we apply the Arnoldi method to some matrix A we obtain the set of vectors $v_1, v_2, \ldots, v_n$ which form an orthonormal basis of the Krylov subspace given by

$$K_n = span\{b, Ab, \ldots, A^{n-1}b\}.$$

This can be seen from the earlier section in this chapter about Arnoldi's method. The general idea is to then find is to approximate the solution to some linear system $Ax = b$ by finding $x_n \in K_n$ such that the residual $r_n = Ax - b$ is minimised. Note that when we apply the Arnoldi method we obtain the matrix V_n made up from the vectors $v_1, v_2, \ldots, v_n$, hence the vector x_n that we

wish to compute can be written as $x_n = V_n y_n$ where y_n in a vector in $\mathbb{R}^n$.

When we apply the Arnoldi method we also produce an upper Hessenberg matrix such that $AV_n = V_{n+1}H_n$. So we can write the following relation

$$||Ax_n - b|| = ||H_n y_n - V_{n+1}^T b|| = ||H_n y_n - \alpha e_1||.$$

Here e_1 is the first vector in the standard basis of $\mathbb{R}^{n+1}$, $e_1 = (1, 0, 0, \ldots, 0)^T$, and for simplicity we denote

$$\alpha = ||b - Ax_0||,$$

where x_0 is some initial guess; it is usually wise to choose $x_0 = 0$.

Hence, we can find x_n by minimising the following linear least squares problem

$$r_n = H_n y_n - \alpha e_1.$$

Solving the linear least squares problem to compute y_n

In order to solve the linear least squares problem, we need to make use of QR factorisation, that is we can decompose some $m \times n$ matrix into an orthogonal matrix Q and an upper triangular matrix R. With this knowledge we can now use the outlined defined in [13] to solve for y_n and then go on to solve for x_n. During the process of solving for y_n we need to make use of Givens rotation [4], which is defined below

Definition 1.51. (Givens Rotation)

A Givens rotation represented by a matrix that takes the form:

$$G = \begin{pmatrix} 1 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \ldots & c & \ldots & -s & \ldots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \ldots & s & \ldots & c & \ldots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & \ldots & 0 & \ldots & 1 \end{pmatrix}$$

Here, $s = \sin(\theta)$ and $c = \cos(\theta)$. s and c appear at the intersections of each row and column. i.e for $i > j$ the non zero elements of the Givens rotation matrix are defined as follows

$$\begin{aligned} g_{kk} &= 1 \text{ for } k \neq i, j \\ g_{kk} &= c = \cos(\theta) \text{ for } k = i, j \\ g_{ji} &= -g_{ij} = s = \sin(\theta). \end{aligned}$$

When we multiply the Given matrix G with some vector, say $\boldsymbol{v}$ then we obtain $G\boldsymbol{v}$ which represents a counter-clockwise rotation of $\boldsymbol{v}$ in the (i, j) plane of θ radians.

We can now continue with solving the linear least squares problem that is to find some y_n that minimises $||H_n y_n - \alpha e_1||$. We start by decomposing our H_n which is an $(n+1) \times n$ matrix by use of QR-factorisation so that we obtain

$$Q_n H_n = R_n.$$

Here Q_n is an orthogonal $(n+1) \times (n+1)$ matrix and R_n is an $(n+1) \times n$ upper triangular matrix. We choose R_n so that it has a bottom row consisting of only zeros, meaning R_n is composed as follows:

$$R_n = \begin{pmatrix} \tilde{R_n} \\ \mathbf{0} \end{pmatrix}$$

where $\tilde{R_n}$ is an $n \times n$ matrix. Now by pre-multiplying the Hessenberg matrix H_n with Q_n we obtain the following:

$$\begin{pmatrix} Q_n & 0 \\ 0 & 1 \end{pmatrix} H_{n+1} = \begin{pmatrix} \tilde{R_n} & r_{n+1} \\ 0 & \beta \\ 0 & \rho \end{pmatrix}.$$

Note that the resulting matrix is almost triangular, in fact it would be triangular if $\rho = 0$ which is where we use the previously defined Givens rotation defined by:

$$G_n = \begin{pmatrix} I_n & 0 & 0 \\ 0 & c_n & s_n \\ 0 & -s_n & c_n \end{pmatrix}$$

where c_n and s_n are calculated by the following formulas:

$$c_n = \frac{\beta}{\sqrt{\beta^2 + \rho^2}} \quad \text{and} \quad s_n = \frac{\rho}{\sqrt{\rho^2 + \beta^2}},$$

which then yields the following results (note here that we do indeed now form a triangular matrix):

$$Q_{n+1} H_{n+1} = \begin{pmatrix} R_n & r_{n+1} \\ 0 & r_{n+1,n+1} \\ 0 & 0 \end{pmatrix}$$

where by our Givens rotation we have $r_{n+1,n+1} = \sqrt{\beta^2 + \rho^2}$. This completes the QR-factorisation leaving us with a triangular matrix and form here on in the minimization problem is relatively simple to solve.

Denote

$$\alpha Q_n e_1 = g_n = \begin{pmatrix} \tilde{g_n} \\ \gamma_n \end{pmatrix}$$

where $\tilde{g_n} \in \mathbb{R}^n$ and $\gamma_n \in \mathbb{R}$. Then consider the following relation:

$$||H_n y_n - \alpha e_1|| = ||R_n y_n - \alpha Q_n e_1|| = \left|\left| \begin{pmatrix} \tilde{R_n} \\ 0 \end{pmatrix} y_n - \begin{pmatrix} \tilde{g_n} \\ \gamma_n \end{pmatrix} \right|\right|.$$

Some simple rearrangement now yields that the vector y_n that minimizes the above equation, is given by the following:

$$y_n = \tilde{R_n^{-1}} \tilde{g_n}.$$

This concludes this section.

In the sequel, we will present the algorithm, along with how to calculate x_n.

GMRES algorithm

The algorithm itself is quite short, details of how to carry out each of the steps in the algorithm are provided earlier in this section. Note that we will not describe the Arnoldi process again, this can be found earlier in the chapter. The algorithm given below is an adaptation of that found in [11] and [13]. We start with some system, say $Ax = b$ then the algorithm is as follows. Note that this algorithm is at the n^{th} iteration:

1. Apply the Arnoldi method (Given earlier in this chapter) to obtain the orthonormal vector v_n.
2. Apply QR-factorisation and solve for the y_n that minimizes $||r_n||$.
3. Compute $x_n = V_n y_n$, if r_n is smaller than tolerance then stop; we have an optimal solution. Otherwise return to step one and repeat until the desired tolerance is met.

1.6 Conjugate gradient method

The conjugate gradient method, originally proposed in [5] by Magnus R. Hestenes and Eduard Stiefel is an iterative method used to solve large linear systems, that are often sparse.

First let us define what it means to have two conjugate vectors, since this is important in this particular method. An interesting property of conjugate vectors is that they are linearly independent; we will show this later in the section.

Definition 1.61.

Consider two vectors u and v. We say that u and v are conjugate vectors with respect to Q if the following holds true. Note that Q is some orthogonal vector.

$$u^T Q v = 0.$$

Definition 1.62.

Let $A \in \mathbb{R}^{n \times n}$ be an $n \times n$ matrix. Then A is said to be positive definite if $x^T A x > 0$ for all non-zero vectors x. Here x is a column vector.

Theorem 1.63.

If Q is a positive definite matrix, and $V = \{v_0, v_1, \dots v_n\}$ is a set of non zero vectors orthogonal with respect to Q then $v_0, v_1, \dots, v_n$ are linearly independent.

Proof:

We will use a proof by contradiction in order to prove Theorem 1.62. We start by saying that if the vectors $v_0, v_1, \dots, v_n$ are not linearly independent

(linearly dependent) then there exists $\alpha_i,\ i = 0, 1, \ldots, n$, with at least one of them non-zero, such that:

$$\alpha_0 v_0 + \alpha_1 v_1 + \ldots + \alpha_n v_n = 0.$$

Meaning that $\alpha_i v_i^T Q v_i = 0$. However we have a contradiction here because Q is in fact positive definite, therefore by definition $v_i^T Q v_i > 0$ which implies that $\alpha_i = 0 \ \ \forall i = 0, 1, \ldots, n$. Therefore, the vectors $v_1, v_2, \ldots v_n$ are linearly independent and this completes the proof by contradiction.

Now let us present some details on how we use the method to solve a system of the type $Ax = b$; for this case we require A to be an $n \times n$ symmetric positive definite matrix.

Suppose that we have a conjugate set consisting of n conjugate vectors, i.e. all pairs of vectors satisfy Definition 1.61 and are conjugate. We denote this set as follows

$$S = \{s_1, s_2, \ldots s_n\}.$$

Since S is a conjugate set, we can say that $s_1, s_2, \ldots s_n$ are linearly independent and hence, form a basis in $\mathbb{R}^n$. The solution to the system $Ax = b$, which we will call x^*, can be expressed in this basis as follows:

$$x^* = \sum_{i=1}^{n} a_i s_i, \qquad a_i \in \mathbb{R}.$$

Then we can multiply both sides by $p_k^T A$ to obtain the following:

$$\begin{aligned} s_k^T A x^* &= s_k^T A \sum_{i=1}^{n} a_i s_i \\ &= \sum_{i=1}^{n} \alpha_i s_k^T A s_i, \end{aligned}$$

which yields

$$s_k^T b = \sum_{i=1}^{n} a_i \langle s_k, s_i \rangle_A,$$

and thus,

$$\langle s_k, b \rangle = a_k \langle s_k, s_k \rangle_A,$$

which gives

$$a_k = \frac{\langle s_k, b \rangle}{\langle s_k, s_k \rangle_A}.$$

We now move on to presenting this method as an iterative procedure. The above method, whilst useful, requires us to compute all of the coefficients a_k for our chosen S. We want an iterative method so that we can approximately solve large systems where we simply wouldn't be able to calculate all of the coefficients. To do this we start with some initial guess for the solution say $x^* = x_0$. We usually assume this to be 0 to simplify our problem slightly. We then make use of the fact that the solution to the original problem x^* minimizes the following quadratic function:

$$f(x) = \frac{1}{2} x^T A x - x^T b.$$

So we take our initial s_0 to be the negative of the gradient of f, which is namely, $Ax - b$ at $x_0 = 0$; so we take our initial vector s_0 to be $b - Ax_0$. Now this implies that the other vectors in S will be conjugate to $Ax - b$ which is where the algorithm gets its name.

Note the following is the residual for our problem at the k^{th} step:

$$r_k = b - Ax_k.$$

We use this to determine how good our solution is and when we should stop. Moreover, we set a pre-determined tolerance too.

Let us now present the algorithm for the iterative conjugate gradient method based on the ones found in both [5] and [11].

Conjugate gradient algorithm

1. Compute the following:

$$r_0 = b - Ax_0$$
$$s_0 = r_0.$$

2. Starting at $k = 0$ compute the following:

3. $a_k = \dfrac{r_k^T r_k}{s_k^t A s_k}$.

4. $x_{k+1} = x_k + a_k s_k$.

5. $r_{k+1} = r_k - a_k A s_k$.

6. If $r_k + 1 < tol$, then stop we have the desired solution. Else continue to the following step.

7. Compute $s_{k+1} = r_{k+1} + \rho_k s_k$ where $\rho_k = \dfrac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$.

8. repeat the process until the desires tolerance is reached and the optimal x_{k+1} is found.

9. End.

Chapter 2

Optimization with MATLAB

In this chapter we will demonstrate solving some optimization problems in MATLAB, both linear and non-linear. We will make use of MATLAB's optimization toolbox to solve some of these problems.

We will start by demonstrating how we can solve some basic linear programs in MATLAB.

2.1 Linear optimization with MATLAB

We can make use of MATLAB's inbuilt linear program solver '*linprog*' to solve linear programming problems stated in the following form where the constraints are grouped by inequality and equality

$$\begin{aligned} &\textit{minimize} && f^T x, \\ &\textit{subject to} && Ax \leqslant b, \\ & && A_{eq}x = b_{eq}, \\ & && l_b \leqslant x \leqslant u_b. \end{aligned}$$

It should be noted here that any inequality constraints of the greater than or equal to type must first be converted to the less than or equal type before we can use MATLAB's '*linprog*' solver to solve. Also note, here, that f is the objective function to be minimized.

Let us begin with a simple example where we call the '*linprog*' solver.

Example 2.11.

Consider the function

$$f(\boldsymbol{x}) = 4x_1 - 2x_2 + x_3.$$

Now consider the following minimization problem associated with the function f

$$\begin{aligned} &\textit{minimize} && 4x_1 - 2x_2 + x_3, \\ &\textit{subject to} && x_1 - x_2 + x_3 \leqslant 15, \\ & && 2x_1 + 3x_2 + 6x_3 \leqslant 30, \\ & && 2x_1 + x_2 = 30 \\ & && 0 \leqslant x_1 \leqslant 15, \\ & && 0 \leqslant x_2 \leqslant 10, \\ & && 0 \leqslant x_3 \leqslant 5. \end{aligned}$$

In order to solve this problem with MATLAB we first need to formulate it in such a way that it looks like the form we gave above. So it is clear to see that we have the following vectors and matrices that form our minimization problem

$$f = \begin{pmatrix} 4 \\ -2 \\ 1 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & -1 & 1 \\ 2 & 3 & 6 \end{pmatrix}, \quad b = \begin{pmatrix} 15 \\ 30 \end{pmatrix},$$

$$A_{eq} = (2, 1, 0), \quad b_{eq} = (30), \quad l_b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad u_b = \begin{pmatrix} 15 \\ 10 \\ 5 \end{pmatrix}.$$

In order to solve this problem in MATLAB we simply insert the following code, calling the '*linprog*' solver to solve the problem

```
f=[4,-2,1];
A=[1,-1,1;2,3,6];
b=[15;30];
Aeq=[2,1,0];
beq=[30];
lb=[0;0;0];
ub=[15;10;5];
[x,fval]=linprog(f,A,b,Aeq,beq,lb,ub)
```

running the code, we obtain the following output as a solution to our linear minimization problem

```
Optimization terminated.

x =

   15.0000
    0.0000
    0.0000


fval =

   60.0000
```

So the solution to our problem and the minimum value that the function f can take is 60, at the point $(x_1, x_2, x_3) = (15, 0, 0)$.

Example 2.12.

We will now solve a similar linear program, but this time we will use the MATLAB Optimization Toolbox; an inbuilt GUI is

for solving all types of optimization problems. Consider the function f where

$$f(\boldsymbol{x}) = -2x_1 - 4x_2 - x_3.$$

Now consider the following optimization problem

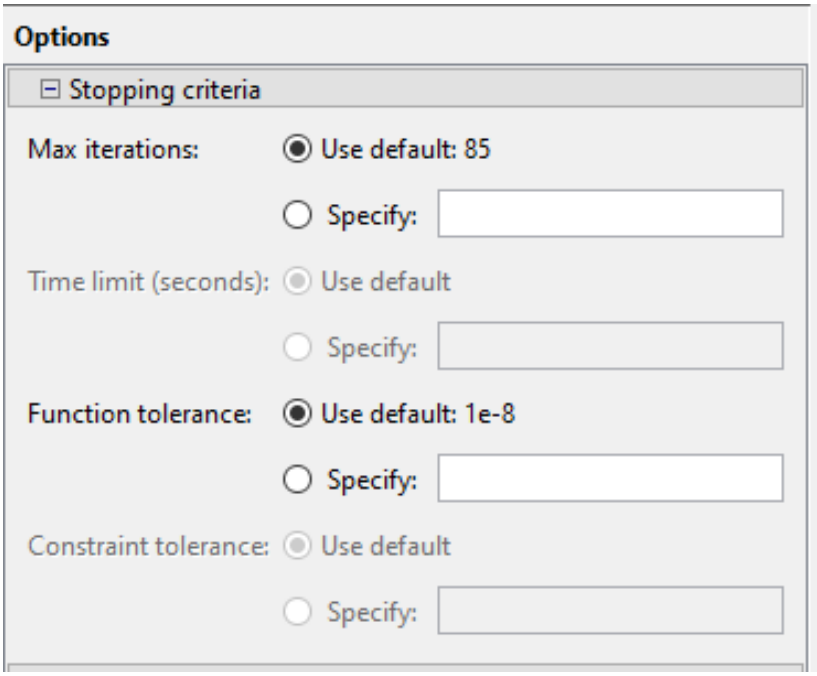
$$\begin{aligned} \textit{minimize} \quad & -2x_1 - 4x_2 - 2x_3, \\ \textit{subject to} \quad & x_1 + x_2 - x_3 \leqslant 15, \\ & 2x_1 + x_2 + 6x_3 \leqslant 30, \\ & 2x_1 + x_2 = 30, \\ & 0 \leqslant x_1 \leqslant 15, \\ & 0 \leqslant x_2, \\ & 0 \leqslant x_3 \leqslant 20. \end{aligned}$$

Note here, that there is no upper bound for x_2. We will simply replace the upper bound here with infinity in MATLAB, so

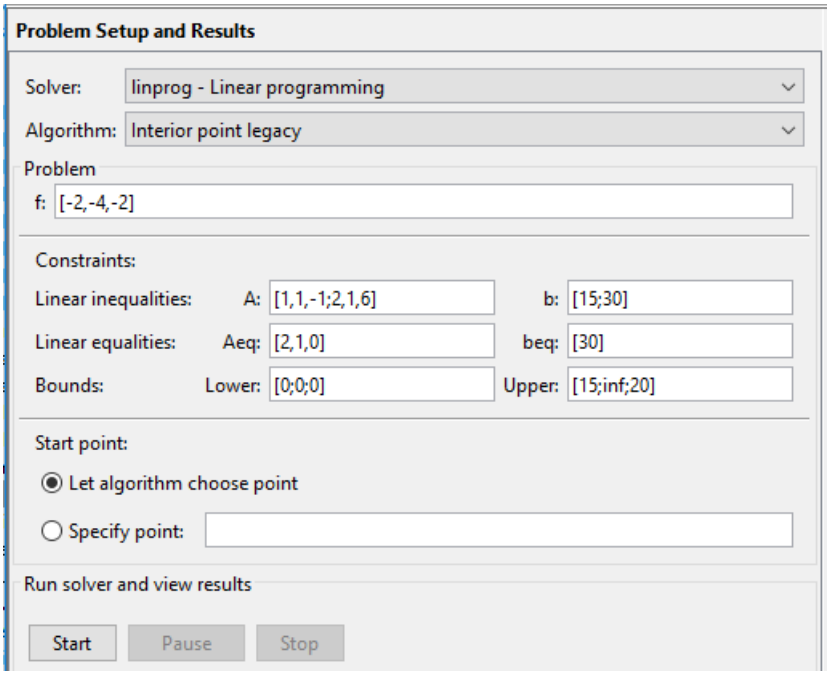
$$u_b = (15, inf, 20)^T.$$

Now let us show how to insert the problem in to MATLAB's optimization toolbox.

We first open the optimization toolbox by simply typing the command '*optimtool*' into the MATLAB command line. Now let us first set our options, in this case we will use the default tolerances.

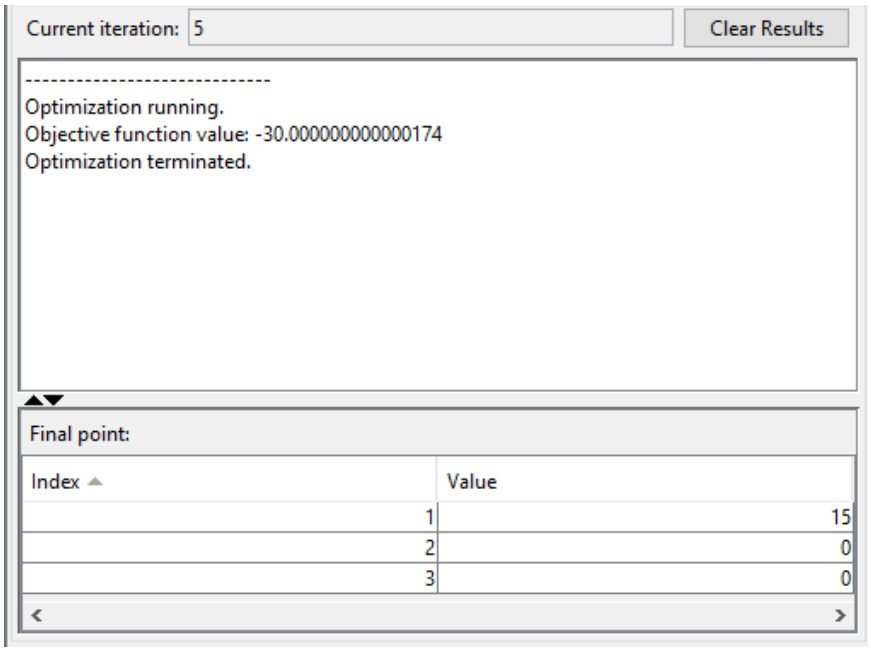


Next we set up the problem, by inserting our vectors and matrices associated with the problem. Note that we have chosen the '*linprog*' solver, so we are using the same routine as in Example 2.10.



The next step is to run the solver. Doing this, we obtain the following solution to this optimization problem, which is -30 at the point $(x_1, x_2, x_3) = (15, 0, 0)$. The solver also presents as output, a message informing that a solution was reached after 5 iterations of the algorithm.

Note that MATLAB defaults to the '*interior point legacy*' algorithm. There is a choice to change this to the '*dual-simplex*' algorithm; both algorithms produce the same results.



2.2 Non-linear optimization with MATLAB

Now that we have looked at some linear examples, we move on to some slightly trickier non-linear optimization problems.

The main difference here is that we will now have to use a different solver and algorithm in order to approximate numerically, the problem's solution

using MATLAB's optimization toolbox. We will need to use various solvers and algorithms depending on the mathematical statement of the problem.

Example 2.21 (The Rosenbrock Function)

A nice non-linear example with constraints for us to try and solve using the optimization toolbox is a problem involving the Rosenbrock function.

Let f be the Rosenbrock function defined by

$$f(\boldsymbol{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

We want to find $\boldsymbol{x} = (x_1, x_2)$ such that we minimise f over the unit disk. So we can write our problem as follows

$$\begin{aligned} &\textit{minimize} \quad 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \\ &\textit{subject to} \quad x_1^2 + x_2^2 \leqslant 1. \end{aligned}$$

The first step to solving this problem in MATLAB is to create an m file for the objective Rosenbrock function. This can be done simply by using the following code.

```
function f = rosenbrock(x)
f=100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

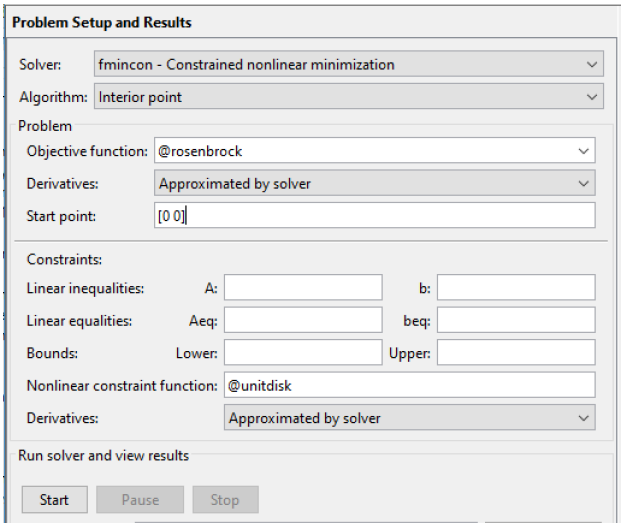
We save this file as 'rosenbrock.m' and we will use it later.

We now need to create another function to define our constraint. Again this can be done simply as follows

```
function [c,ceq] = unitdisk(x)
c = x(1)^2 + x(2)^2 -1;
ceq = [];
```

Here the '*c*' part represents the inequality constraints and the '*ceq*' part represents the equality constraints. This is why we have left this blank on this occasion. We save this file as 'unitdisk.m'.

We will now proceed to the optimization toolbox to solve the problem. We open the optimization toolbox as before by typing '*optimtool*' into the MATLAB command line. Alternatively the toolbox can be opened via the Apps tab on the command bar.



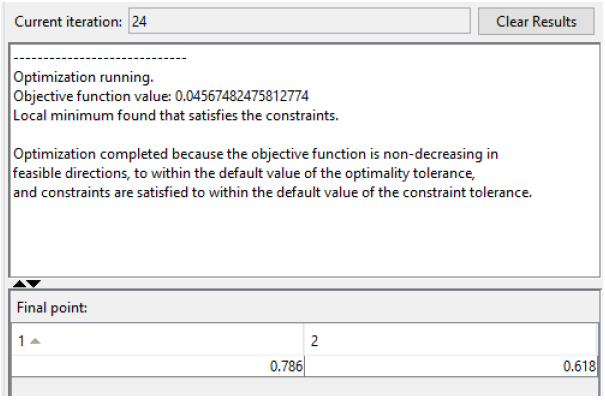
Note here that we use the default solver '*fmincon - Constrained nonlinear optimization*' along with the default algorithm '*Interior point*'. This solver is MATLAB's default and can be used for any linear or non-linear constrained minimization problems.

We call our previously defined objective and constraint functions using an @ sign. We also tell the solver to begin the search for a solution at

$$(x_1, x_2) = (0, 0).$$

We do this because it is obvious from the function that the optimal values will be close to 0.

We then click '*start*' to begin running the solver, which produces the following output



So we have an optimal function value of 0.046 (3 d.p) which is given at the point $(\boldsymbol{x}) = (x_1, x_2) = (0.786, 0.618)$. We also note that the algorithm ran for 24 iterations before the default tolerances were met.

Example 2.22.

We will now look at solving a non-linear optimization problem with no constraints using the optimization toolbox.

We can now just simply state the problem as follows

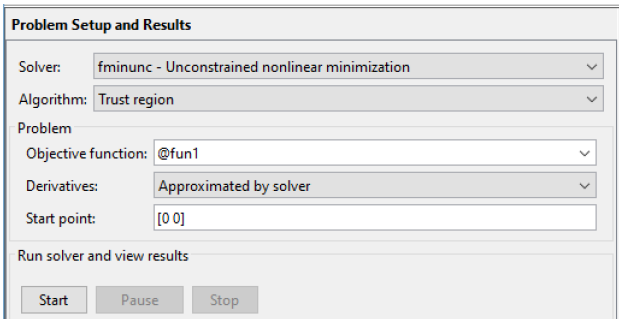
$$\min_{\boldsymbol{x}\in\mathbb{R}^2} f(\boldsymbol{x}), \quad \text{for} \quad f(\boldsymbol{x}) = 2e^{x_1}\left(x_1^2 + 3x_2^2 + 5x_1x_2 + x_2 + 1\right).$$

Again we need to define the function as an m file in MATLAB so that we can call it when we run the solver. We can do this as follows

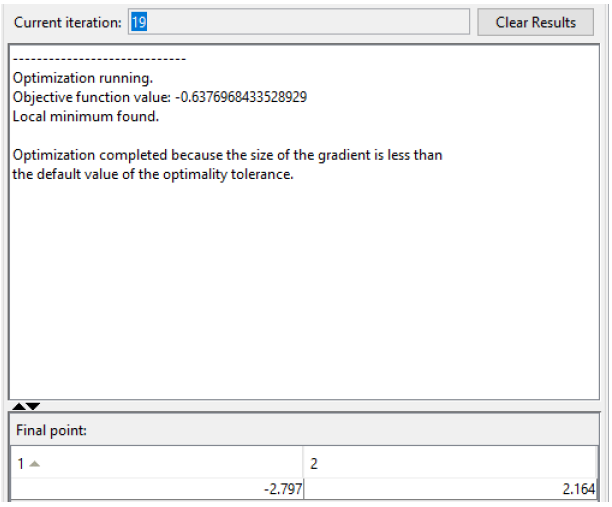
```
1 function f = fun1(x) f =
2 2*exp(x(1))*(x(1)^2+3*x(2)^2+5*x(1)*x(2)+x(2)+1);
```

We save this file as '*fun1.m*', so that we can call it when we use the solver in the optimization toolbox. Since this problem is unconstrained, we do not need to write an m file for the constraints.

We can now skip directly to the optimization toolbox to solve our unconstrained minimization problem. Since we now have an unconstrained non-linear minimization problem we need to choose our solver and algorithm appropriately. The most suitable solver here is the '*fminunc - Unconstrained nonlinear minimization*'.



Note here that because we don't have any constraints then the setup of the problem becomes slightly more simple and we only need to call the objective function '*fun1*' and choose a starting point, which we chose to be $\boldsymbol{x_0} = (0, 0)$. Note also here that we used the default '*Trust region*' algorithm, the other option available to us is the '*Quasi Newton*' algorithm, both produce the same result, we will show this later.

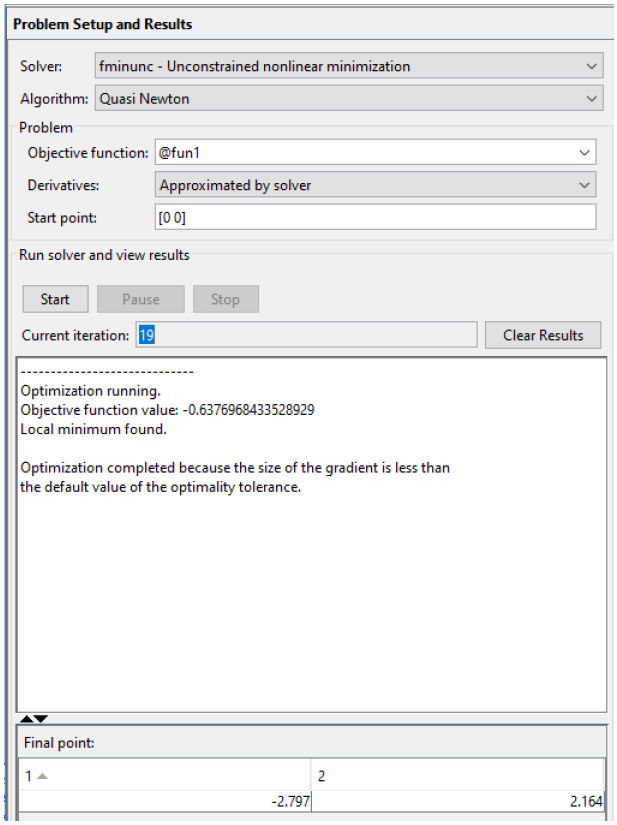


So when using these options we get a solution of -0.638 obtained at the point

$$(\boldsymbol{x}) = (x_1, x_2) = (-2.797, 2.164),$$

all given to 3 d.p.

Now let us show that we do indeed obtain the same result via the alternative algorithm



Note that not only do both algorithms produce the same solutions, they also both solve our problem after 19 iterations.

2.3 Maximization with MATLAB

We can't solve maximization problems directly by using MATLAB, but it is easy to convert such a problem in to a minimization problem and then solve it using the optimization toolbox.

Note that

$$\max_{\boldsymbol{x}} f(\boldsymbol{x}),$$

is identical to

$$\min_{\boldsymbol{x}} -f(\boldsymbol{x}).$$

So we can easily solve maximization problems using the toolbox by simply multiplying the objective function by -1.

Example 2.31.

Consider Example 2.11. that we solved earlier in the chapter. Let's say that we wanted to find the maximum of this function under the same constraints; we would write the problem as follows (also note that we have added an upper

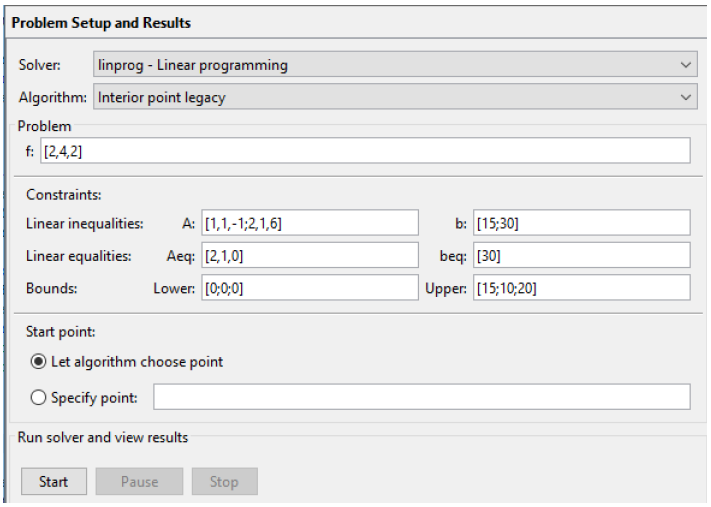
bound to the x_2) coordinate

$$\begin{aligned} \textit{maximize} \quad & -2x_1 - 4x_2 - 2x_3, \\ \textit{subject to} \quad & x_1 + x_2 - x_3 \leqslant 15, \\ & 2x_1 + x_2 + 6x_3 \leqslant 30, \\ & 2x_1 + x_2 = 30, \\ & 0 \leqslant x_1 \leqslant 15, \\ & 0 \leqslant x_2 \leqslant 10, \\ & 0 \leqslant x_3 \leqslant 20. \end{aligned}$$

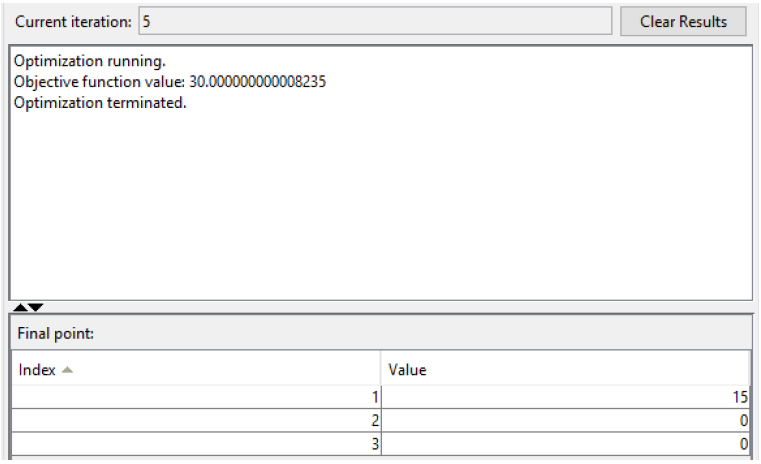
So by multiplying the objective function $f(\boldsymbol{x})$ by -1, we write the equivalent minimization problem as follows, noting that we do not need to change our constraints.

$$\begin{aligned} \textit{minimize} \quad & 2x_1 + 4x_2 + 2x_3, \\ \textit{subject to} \quad & x_1 + x_2 - x_3 \leqslant 15, \\ & 2x_1 + x_2 + 6x_3 \leqslant 30, \\ & 2x_1 + x_2 = 30, \\ & 0 \leqslant x_1 \leqslant 15, \\ & 0 \leqslant x_2 \leqslant 10, \\ & 0 \leqslant x_3 \leqslant 20. \end{aligned}$$

We can now input this problem in to MATLAB's optimization toolbox as follows



Note that here, we have only changed the signs in the objective function. We then run the solver to obtain the following solution



So the solution to our problem is 30, which is obtained at the point

$$(x_1, x_2, x_3) = (15, 0, 0).$$

Chapter 3

The travelling salesman problem

The travelling salesman problem is a classical problem in optimization, first described by Hamilton and Kirkman in the 1800's; it wasn't until the 1930's when the problem was defined and potential solutions were offered the first of these is the most obvious brute force algorithm.

The problem itself centres around a salesman who needs to visit N cities, in any order, finishing at the start city. The salesman wants to be able to achieve this by travelling as little total distance as possible with minimum cost.

The cost is determined by a method of travel between each destination; of course flying being more expensive than say using the train or driving to a destination. This makes the problem particularly difficult because we have N cities along with varying numbers of different travel methods to get from each city to another. Often we ignore the varying methods of travel and assume that the salesman uses the same travel method between each city, eliminating thus, the cost factor of the problem.

Selected Milestones

The table below shows a number of selected milestones in the computation of the travelling salesman problem. The current world record stands at 85,900 nodes, computed in 2006 using the Concorde (`http://www.math.uwaterloo.ca/tsp/concorde/`) program.

Year	Number of Nodes (Cities)	Researcher(s)
1954	49	G. Dantzig, R. Fulkerson and S. Johnson
1971	64	M. Held and R. Karp
1977	120	M. Grotschel
1987	2,392	M. Padberg and G. Rinaldi
2004	24,978	D. Applegate, R.E. Bixby, V. Chvatal and W. J. Cook
2006	85,900	D. Applegate, R.E. Bixby, V. Chvatal and W. J. Cook

3.1 Stating the problem

Since, essentially the travelling salesman problem is a linear (highly non-trivial) optimization problem where we want to minimize the distance travelled.

We can formulate it as a linear program, this is done as follows:

Assume we have N cities labeled 1 to n. Let u_i be a dummy variable (inserted artificially in order to give some extra degrees of freedom to the problem, however resulting to an equivalent problem solvable with the standard Simplex method) and take w_{ij} to be the distance between two cities, i and j. Then define x_{ij} as

$$x_{ij} = \begin{cases} 1, & \text{the path goes from city } i \text{ to city } j, \\ 0, & \text{otherwise.} \end{cases}$$

Then we can state our problem as follows:

$$\textit{minimize} \quad \sum_{i=1}^{n} \sum_{j\neq i,\ j=1}^{n} w_{ij}x_{ij} \tag{3.1}$$

$$\begin{aligned} \textit{subject to} \quad & 0 \leqslant x_{ij} \leqslant 1 && i,j = 1,\dots,n && (3.2)\\ & u_i \in \mathbb{Z} && i = 1,\dots,n && (3.3)\\ & \sum_{i=1,\ i\neq j}^{n} x_{ij} = 1 && j = 1,\dots,n && (3.4)\\ & \sum_{j=1,\ j\neq i}^{n} x_{ij} = 1 && i = 1,\dots,n && (3.5)\\ & u_i - u_j + nx_{ij} \leqslant n-1 && 2 \leqslant i \neq j \leqslant n. && (3.6) \end{aligned}$$

Here, the first inequality requires that each city is only visited once, i.e. each city is arrived at from exactly one other city. The 2 equalities require that each city is departed only once to exactly one other city and the final inequality restricts us to having only one tour that covers all cities, i.e. we cannot have two or more sub tours that cover the cities.

3.2 Methods of solution

There are various methods of solution that have been proposed for the TSP over many years.

Let us start with the obvious one. Brute force, we try all of the options and then pick the optimal one (smallest total distance) at the end. This method has an obvious drawback in that it becomes extremely impractical even for a very small amount of cities. This is because for N cities the number of possible routes is given by

$$\text{No. of Possible Routes } = \frac{(N-1)!}{2}.$$

This makes the brute force method impractical for any more than 5 cities if you do it by hand.

For example if we had $N = 20$ cities then there are $60,822,550,204,416,000$ possible routes that we can take. Using modern computing power, to determine all the possible options of this order, it isn't so difficult, however it costs and it is very inefficient compared to other methods of solution.

For this reason there are many approximation algorithms that exist for solving the TSP. We will look at just a couple of these in the next paragraph.

Approximation algorithms are just that, approximations, therefore we cannot guarantee that we will reach an optimal solution. Most approximation algorithms will find a 'good solution', often within 1.5 times the optimal one. Some of these algorithms are listed below:

- Branch & Bound Algorithms - Many of these exist and do find the optimal solution, they are the most widely used methods for solving the TSP with around 40-200 cities [3].
- Ant Colony Optimization - A method that can generate 'good solutions' that uses real data collected from ant colonies taking into account the routes they take when they build nests and find food. This is then converted into a computer program that releases virtual ants that test options based on ant behaviour in real life [2].

3.3 An Example with 4 cities

Let us now consider a rather simple example with only 4 cities or nodes, these nodes can be represented on the following graph in Figure 3.1. Note that the graph is not to scale. We have labelled the cities A, B, C and D. The distances between each node are labelled on the graph, these distances don't have any units but could be anything. In the case of a salesman travelling by car, they would most likely be kilometres or miles. Our starting point is A so we will start and end at A.

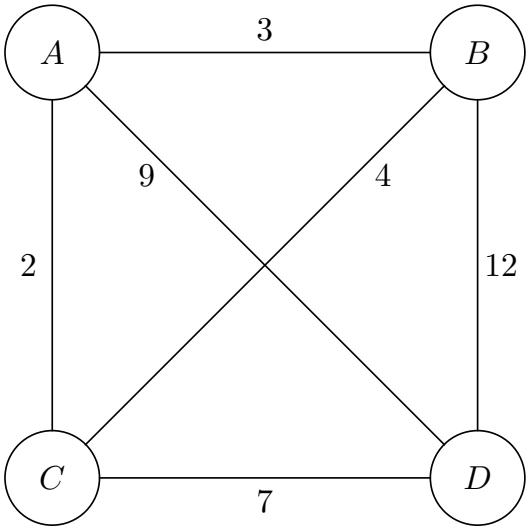
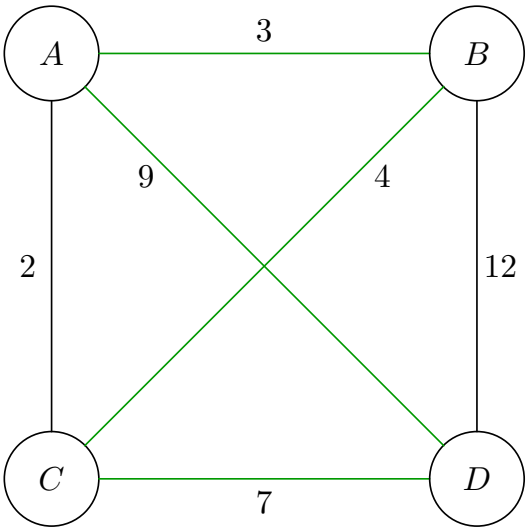
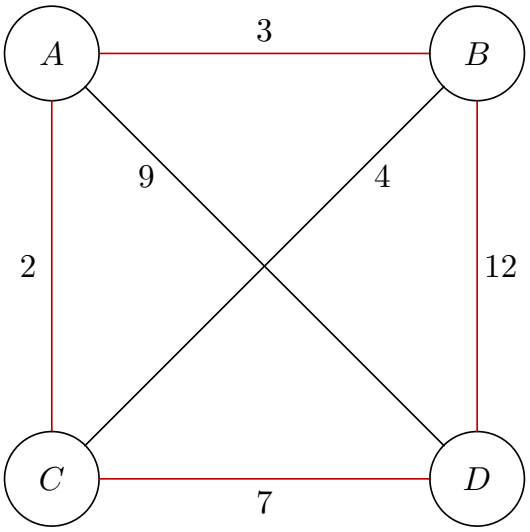


Figure 3.1: A graph showing a network consisting of 4 cities.

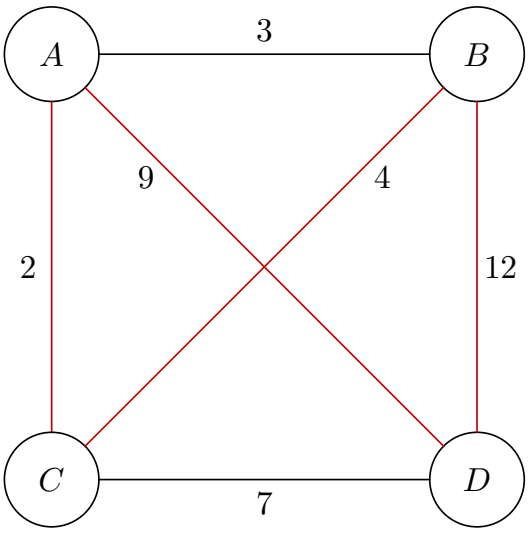
We can calculate the number of possible routes easily using the formula from section 3.1, doing this we find the the number of possible routes on our network is given by $(4-1)!/2 = 3$. So using a brute force approach to solving this problem we need to calculate the distance on each of these 3 routes and then



(a) $A \to B \to C \to D \to A = 23$.



(b) $A \to B \to D \to C \to A = 24$.



(c) $A \to C \to B \to D \to A = 27$.

Figure 3.2: Graphs showing all three of the possible routes around our network visiting all 4 nodes starting and ending at the point A.

choose the optimal route, which in this case is of course the minimum. The 3 possible routes are shown below, with the optimal one highlighted in green and the remaining non-optimal routes highlighted in red.

- $\boldsymbol{A \to B \to C \to D = 23}$
- $\boldsymbol{A \to B \to D \to C \to A = 24}$
- $\boldsymbol{A \to C \to B \to D \to A = 27}$

All three routes are plotted on the graphs in Figure 3.2 again the optimal route is highlighted in green and the non-optimal routes in red. Note from Figure 3.2a that we actually miss out the shortest journey from A to C of 2 units in our optimal solution.

So it can be seen in Figure 3.2 that our optimal route around the 4 cities is $A \to B \to C \to D \to A$ and is of length 23, 1 unit shorter than the next best route seen in Figure 3.2b and 4 units shorter than the least optimal route, seen in Figure 3.2c. Here we used a brute force search to find our optimal solution, this works well and can be done by hand when we have 5 or less nodes, however it becomes much more difficult as N increases to anything larger than that and we need to make use of a computer to solve our problem. In the next example we will see this as we increase N to be 200.

3.4 An Example with 200 cities

A good way of visualizing a solution to the TSP is to compute a solution in MATLAB, we can use the code from `https://uk.mathworks.com/help/optim/ug/travelling-salesman-problem.html`. The code draws a map of the United States of America and then randomly plots $N = 200$ cities. This plot containing our 200 cities can be seen below in Figure 3.3. Notice that this is much more complex than the example we gave in the previous section and we would never attempt to solve a problem like this without a computer program, as it would just take way too long.

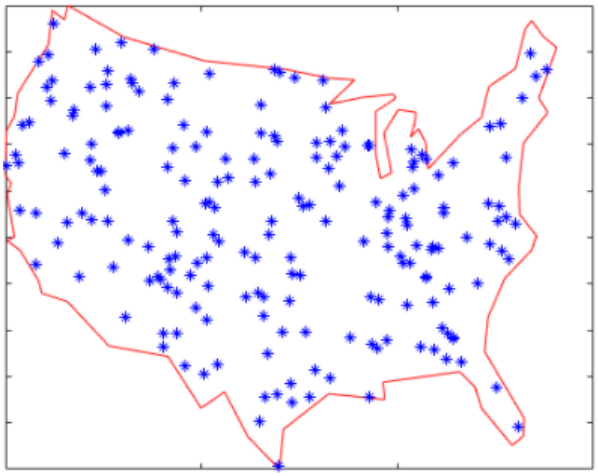


Figure 3.3: Border of the USA, with 200 randomly plotted cities.

We then run the code referenced earlier, noting that this code will first find a solution that includes sub-tours and then eliminates them, we do not want sub-tours because then there will be more than one city which has been visited twice, we only want one city to be visited twice, the starting point, because we start here and want to finish here too. The solution with sub-tours can be seen in Figure 3.4. We do not want a solution with sub-tours though so we will eliminate them in the next stage.

Now we mentioned that we do not want any sub-tours in our solution, i.e. we want to visit each of our 200 cities once and only once, except of course the starting and ending city, which we visit twice, once when we start there and then again at the end when we arrive back. The code referenced earlier in this section first solves the problem allowing sub-tours. It then re-applies the

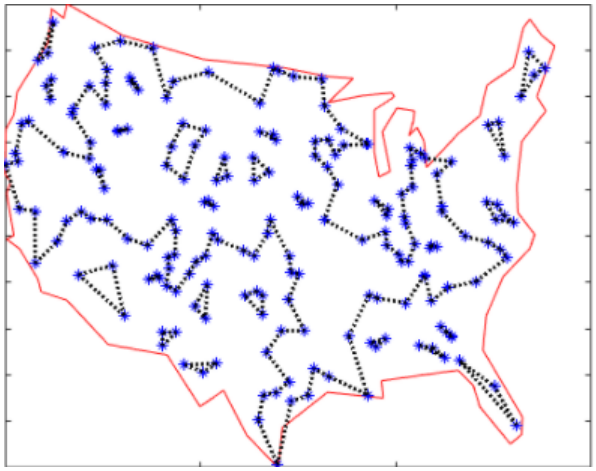


Figure 3.4: Solution to the TSP with $N = 200$, containing sub-tours

algorithm until it can no longer detect any sub-tours. At this point we are happy that we have the final solution to our travelling salesman problem that doesn't include any sub-tours.Such a solution can be seen in Figure 3.5.

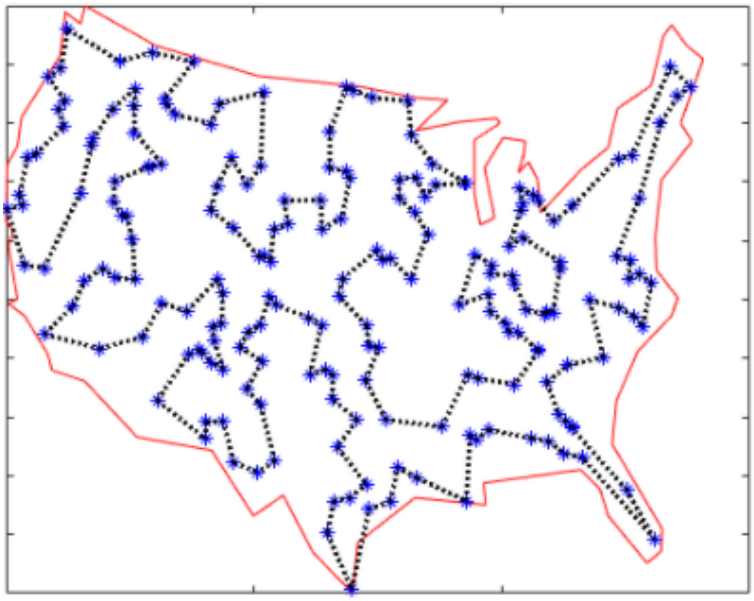


Figure 3.5: Solution to the TSP with $N = 200$, with sub-tours eliminated.

It can now be seen that in Figure 3.5 we have one continuous journey that starts and ends in the same place that visits each of the other 199 cities exactly once. This is the optimal solution to this particular problem. Note that we cannot give a numerical distance here because the problem itself is not scaled.

Chapter 4

Monte Carlo methods

Monte Carlo methods is a family of algorithms applied for the numerical approximation of problems in the presence of randomness, [14]. Randomness appears in physical, biological or financial problems in various cases, as for example due to initial conditions that are not exact (initial data noise due to measurement error), or thermal fluctuations during a physical experiment, or even due to the Heisenberg uncertainty principle of elementary physical particles. Moreover, when a very large amount of information, the so-called big data, is involved in the statement of a problem (statistical mechanics, computational physics, material science, or financial mathematics) Monte Carlo methods are used in combination with statistical methods.

In Monte Carlo methods two basic strategies are applied: 1) randomness is inserted by the use of certain distributions implemented for example in Matlab, and 2) the execution of the same algorithm many times (usually upwards of 1000 times) with randomly chosen initial data. The numerical solution considered, is the mean value of the collected output data, while the error of the numerical method applied, is estimated in expectation or other probabilistic norms.

4.1 A simple dice rolling experiment

Suppose we have two dice, and we want to experimentally compute the probability that when we roll both dice together they sum to a given value. We could of course do this by hand, rolling the dice say 100 times and recording our results and then the final probability will be given by the number of times the dice sum up to our desired value, divided by the total number of runs we did, 100.

How accurate will this answer be, are the dice ideal, or biased?

Even in the unbiased case, theoretically, according to the Central Limit Theorem of probability, for obtaining the accurate answer to the previous question, we need an infinite time of trials, in order to specify if the dice are ideal.

For the purpose of our experiment let's say we have two (here, assumed) unbiased six sided dice, and we want to know the probability of obtaining a score of 7 when we add up the values of both dice when rolled together.

Of course we can work this theoretically quite easily, as there are 6 ways of obtaining a 7 and 36 total possibilities, the answer is simply $\frac{6}{36} \approx 0.167$. But we want to use a Monte Carlo approach to find this probability experimentally. To save us rolling the dice over and over again we can build a MATLAB program

to simulate this experiment for us. This is done using the following code:

```
%Number of Dice, Faces & Number of Rolls
S = 2;
F = 6;
N = 100;
%Roll Dice and take sum of both rolls
Rolls = randi(F, N, S);
SumRolls = sum(Rolls, 2);
%Possible outcomes, for histogram bars
Outcomes = (S:F * S)';

%Histogram of results
hist(SumRolls, Outcomes);
title('Histogram of results');
xlabel('Sum of both rolls');
ylabel('Count');
%Add labels to each bar
[n,x] = hist(SumRolls, Outcomes);
barstrings = num2str(n');
text(x,n,barstrings,'horizontalalignment','center',
    'verticalalignment','bottom')
```

Since 100 rolls seems a suitable amount of experiments to do by hand let us start with $N = 100$.

Doing so we get the following result shown on a histogram plot:

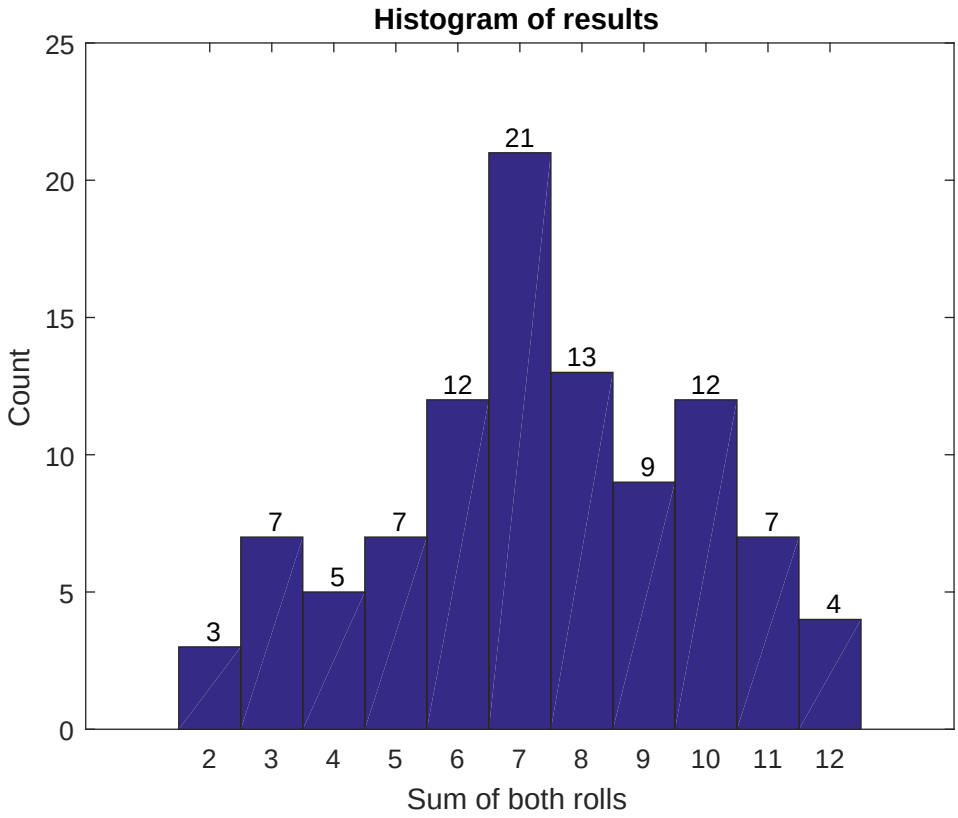


Figure 4.1: Histogram of results with $N = 100$.

So with 100 trials, we get that the probability of obtaining a combined score of 7 is given by $\frac{21}{100} = 0.21$.

This is a little higher than expected, in fact calculating the error in this estimation, we find that we have approximately %25 error in this particular experiment. This is a very large error and indicates that 100 runs is much too small for this experiment.

Let us try again, this time doubling the number of experiments to $N = 200$. We obtain then the following histogram plot of results:

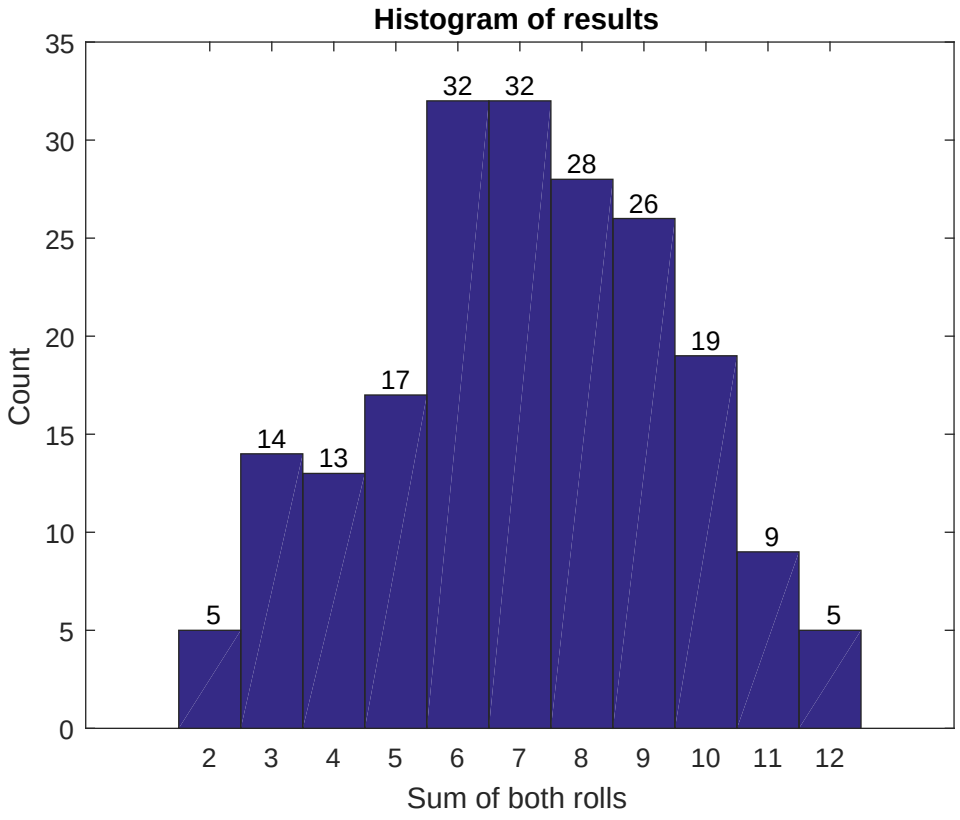


Figure 4.2: Histogram of results with $N = 200$.

It can be observed in Figure 2 that we are starting to see a distribution of the results near to what we expect. We can now calculate the probability of obtaining a combined score of 7 to be 0.16, giving us a % error of approximately just 4%. So, by doubling our number of runs, we have considerably improved our result, but how many runs will we need to obtain an even better result.

Let us try the experiment with $N = 1000$. The histogram of results can be seen below in Figure 3.

It can be seen that we now have a very distinctive normal curve appearing in our results profile. We would expect the distribution to look like this based on the probabilities of obtaining each result. Calculating the probability of obtaining a combined score of 7 from our histogram we now find this to be 0.169, leaving us now with a % error of just 1.2%.

The table below show how the error decays as we increase N, the number of runs.

It can be seen from Table 4.1 that when we increase N we improve the error in the estimation; obviously this is expected. Notice also from these results that for a good approximation we ideally need to choose N very large (at least

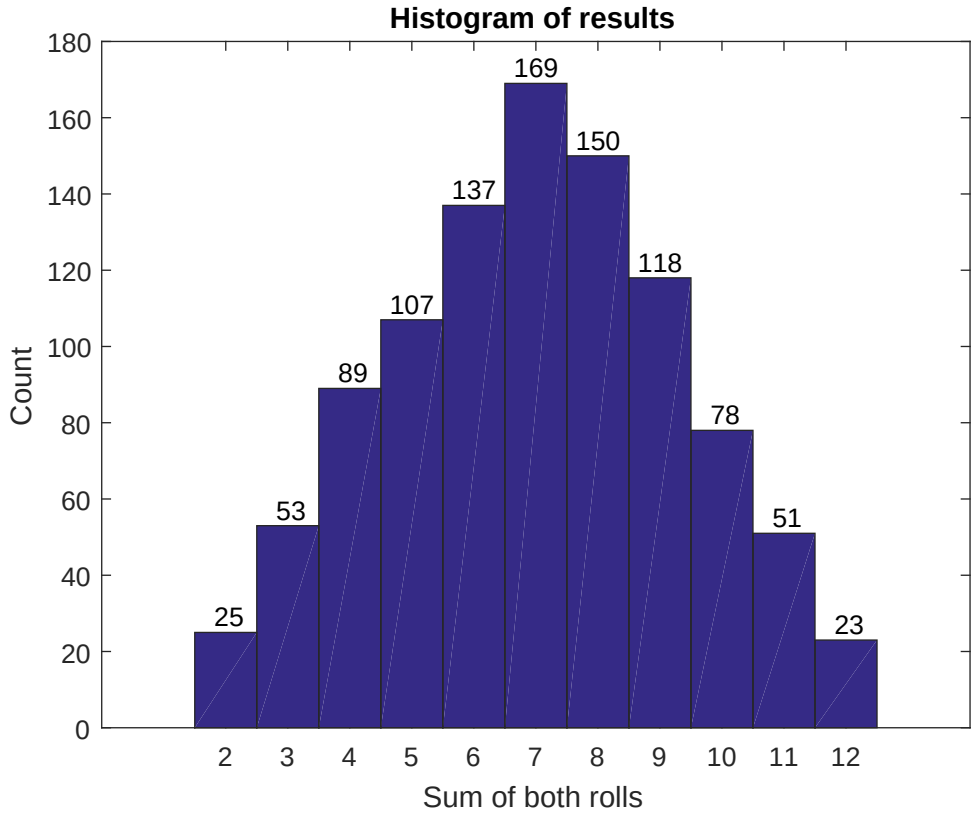


Figure 4.3: Histogram of results with $N = 1000$.

N	No. of times combined score of 7 is obtained	Probability	% Error
100	21	0.2100	25.0
200	32	0.1600	4.19
1000	169	0.1690	1.20
10000	1683	0.1689	1.14
100000	16616	0.1662	0.48

Table 4.1: Table of results from dice rolling experiment.

1000). This of course depends on how accurate we want the approximation to be.

4.2 Applying Monte Carlo to an ODE

We can apply a Monte Carlo approach to an ODE's numerical approximation in the presence of randomness.

Consider the following example related to a harmonic pendulum.

Example 4.21. (Harmonic pendulum: deterministic case)

Consider the following second order ODE, for a harmonic pendulum of frequency ω, with $\boldsymbol{x}$ defining the coordinates of the ideal nodal mass pendulum in dimensions 2, i.e. we consider the motion on the plane, in the absence of

friction

$$\ddot{\boldsymbol{x}} + \omega^2 \boldsymbol{x} = 0, \qquad \omega > 0.$$

We will take first $\omega = 1$; then converting this in to a system of ODE's we have the following

$$\dot{x} = y,$$
$$\dot{y} = -x,$$

which has an equilibrium point at $(0, 0)$.

This system of ODE's can be implemented as a function in MATLAB as follows:

```
function ex1 = ex1(t, y)
ex1 = [y(2), -y(1)]';
end
```

We then use the following MATLAB code to solve our problem, firstly without any randomness; we will introduce this later.

We use the MATLAB's ODE45 solver, which is a mixture of fourth and fifth order Runge-Kutta methods.

Here, we solve numerically our ODE for various different initial conditions at time $t = 0$; our numerical results show that the orbits are all centred at the equilibrium point $(0, 0)$.

```
format long e

%%% Set ODE45 Options %%%
options = odeset('RelTol', 1e-4,'AbsTol', [1e-4 1e-4]);

%%%Graph & Axis titles%%%
title('orbits')
xlabel('x')
ylabel('y')

hold on
for k = 1 : 20
[t, u] = ode45('ex1', [0, 100], [0.1*k, 0.1*k], options); ...
    %Call ODE45
plot(u(:, 1), u(:, 2)) %Plot Phase Plane
end
hold off
```

This produces the following phase plane plot of the orbits. We can see in Figure 4.5 that all of the orbits are centred at the equilibrium point (0,0) as we would expect.

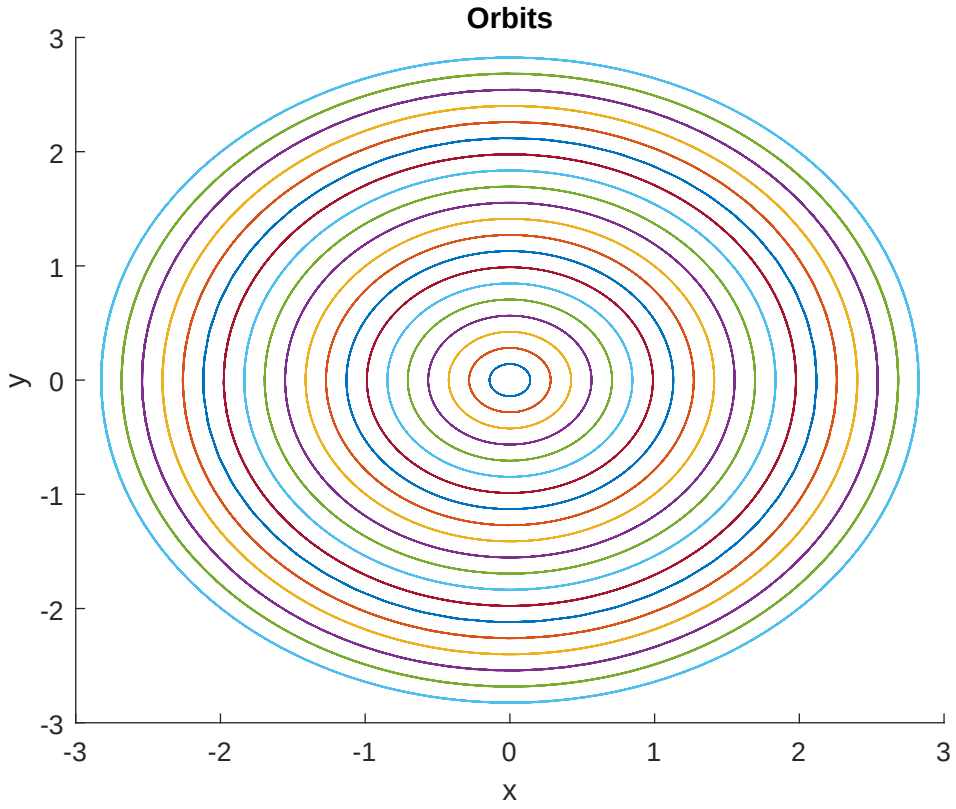


Figure 4.4: Phase Plane for Harmonic Pendulum.

Changing the tolerance of the solver

In the previous experiment we had tolerances of 10^{-5}. Let us now increase this to 10^{-2}. This means that now the '*ODE45*' will now solve our ODE with less accuracy. The effect that this has on our orbits in the phase plane can be seen in the next figure. We observed that we now have jagged ellipses; this is because the solver hasn't solved the ODE with enough accuracy to ensure a smooth solution.

Changing the value of ω

In the previous example we chose to use $\omega = 1$. However we are interested to find out what happens to our orbits when we use different values of ω. We start with $\omega = 0.5$. Note that again we run the experiment with 20 different initial conditions as we did previously in the example where we had $\omega = 1$.

It can be seen from Figure 4.6 that when we have $\omega = 0.5$ the orbits are stretched in the x direction meaning that the circles we saw in the previous example are now more ellipses that are more oval shaped. Let us next see what happens when we further decrease ω to $\omega = \frac{1}{3}$.

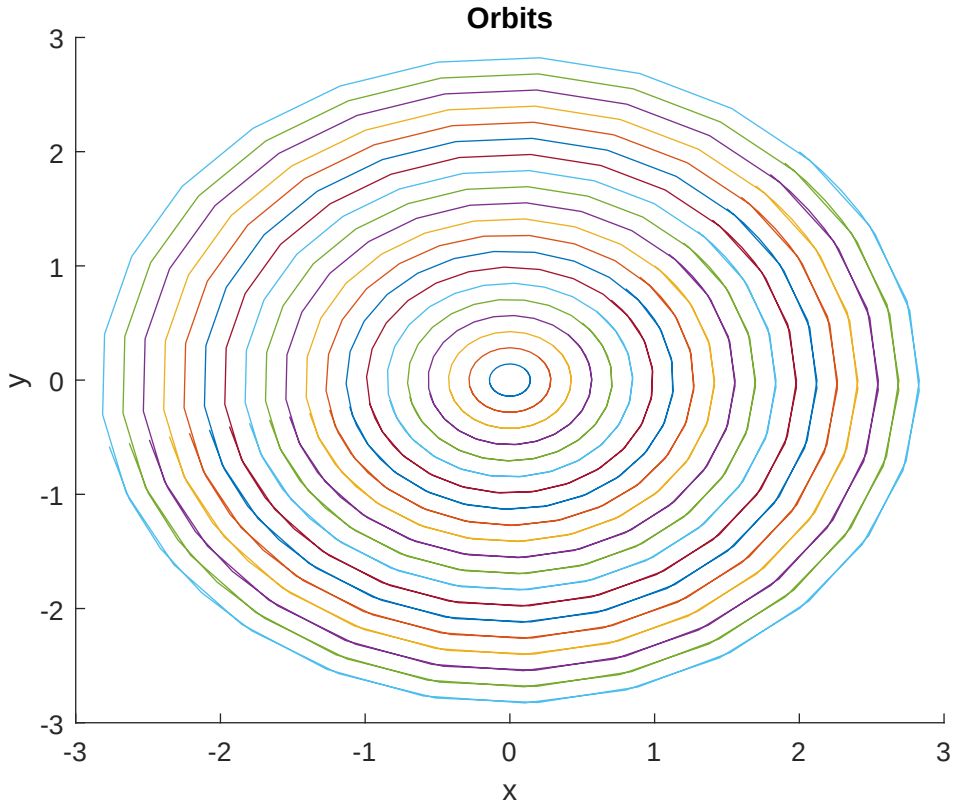


Figure 4.5: Phase Plane for Harmonic Pendulum with tolerance of 10^{-2}.

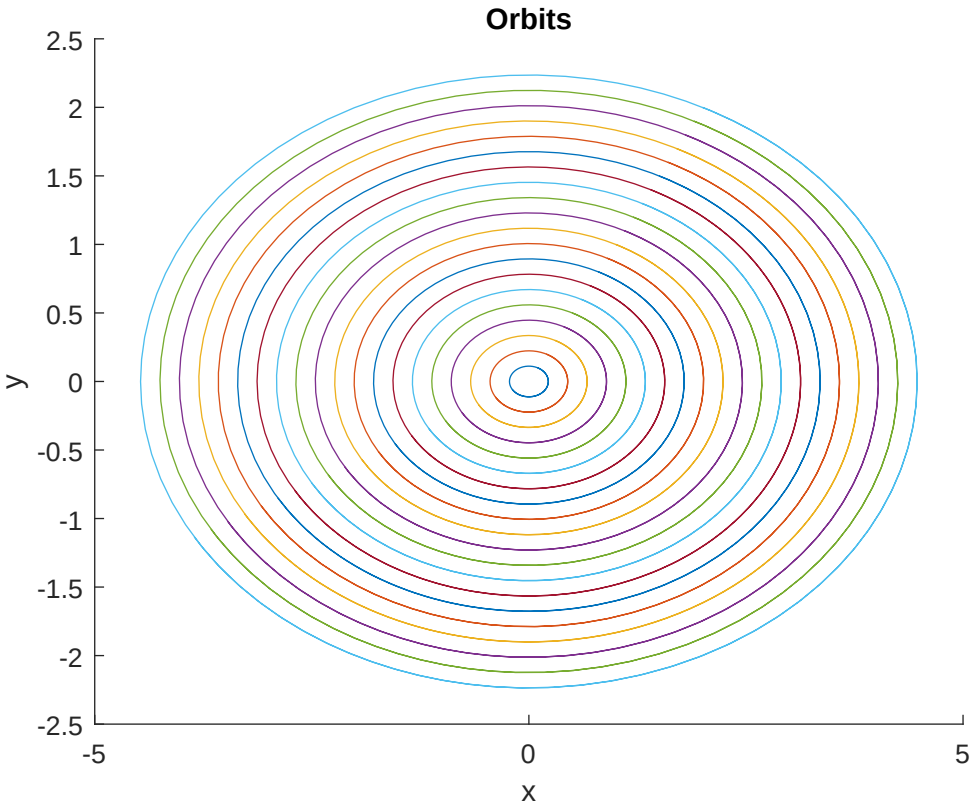


Figure 4.6: Phase Plane for Harmonic Pendulum with $\omega = 0.5$.

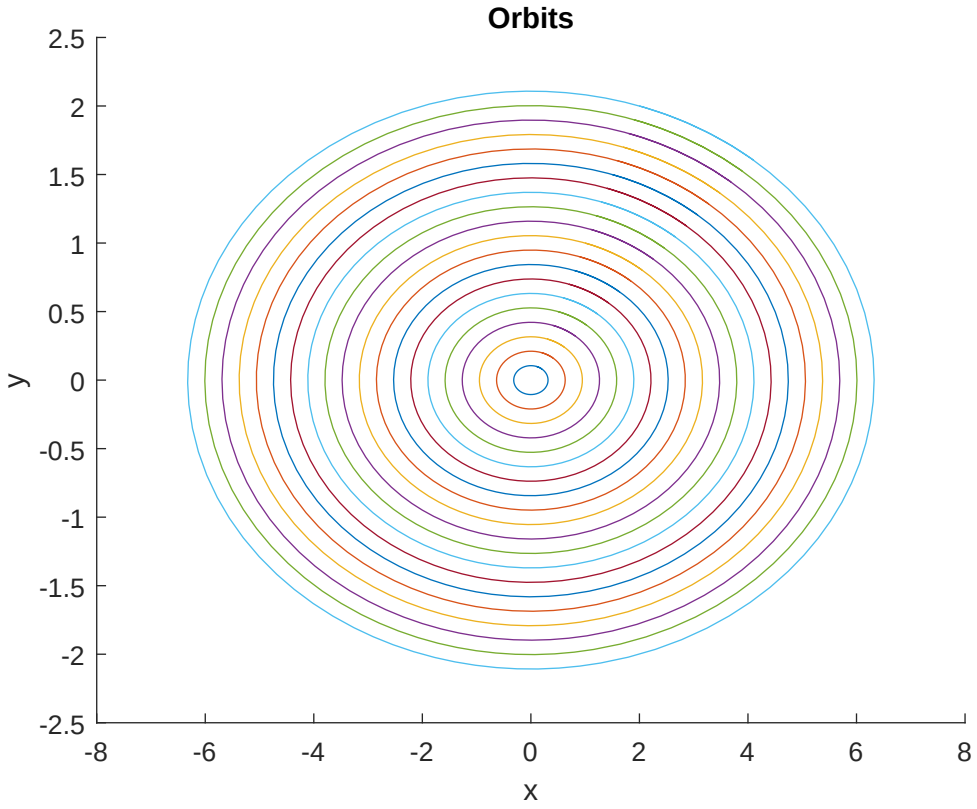


Figure 4.7: Phase Plane for Harmonic Pendulum with $\omega = \frac{1}{3}$.

We can see again from Figure 4.7 that that the ellipses have been stretched in the x direction. Notice that the y direction remains unchanged, this is because y is independent of ω.

So, our numerical results verified numerically, some theoretical properties of this pendulum (ODE): periodic solution i.e. the so-called center equilibrium point (closed orbits after finite time), and elliptic orbits with axes being the 2-dimensional eigenvector basis produced by the ODE's matrix eigenvalues.

Definition 4.22. (Brownian Motion)

Named after English botanist Robert Brown, Brownian motion describes the zig-zagging motion of some small particle when it immersed in a liquid or a gas [10].

A useful property of Brownian motion is that it follows the normal distribution i.e. $B(t) \sim \mathcal{N}(0, t)$. Thus, the Brownian motion has probability density function

$$f_{B_t}(x) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}}.$$

Note that the mean value of B_t is

$$E[B_t] = 0$$

and the variance is given by

$$\text{var}(B_t) = t.$$

We introduce the notation

$$B(t) := B_t.$$

B_t is an one dimensional stochastic process, depending on $t \in [0, \infty)$, while according to the Normal distribution scaling

$$E(aB_t) = 0, \quad \text{var}(aB_t) = a^2 t,$$

for any real a, since $aB_t \sim \mathcal{N}(a \cdot 0, a^2 t) = N(0, a^2 t)$.

We will use the definition of Brownian motion to introduce randomness into our ODE, this can be seen in the next set of experiments.

Now that we have solved the problem without any randomness we want to introduce some randomness into the system. Let us first introduce randomness into the system itself as an additive stochastic force, i.e. in ODE terminology, a stochastic non-homogeneous term.

We can do this with the following system of differential equations.

$$\dot{x} = y + 10^{-3} \times B(1)$$
$$\dot{y} = -x + 10^{-3} \times B(1)$$

Here $B(1)$ represents the one dimensional Brownian motion at $t = 1$. Multiplying our random values by 10^{-3} ensures that the changes in the system are small. Larger changes could supplement the system with severe numerical instability. Note that due to linearity, theoretically any such system has a solution, since the homogeneous one has. However, in non-linear problems, the additive forcing, stochastic or not, can result to systems of ODEs where existence of solution is not satisfied. Thus the strength, here of randomness (in our case, the variance 10^{6t}) must be sufficiently bounded in $t \in [0, T]$, for T the final time of experiment.

We implement this system in MATLAB as follows using the '*normrnd*' function in MATLAB to produce our random values. We save this function as '*ex5d.m*' so that we can call it from ODE45 later.

```
function ex5d = ex5d(t, y)

f1=10^(-3)*normrnd(0,1); %Random values from N(0,1)
f2=10^(-3)*normrnd(0,1);

ex5d = [y(2)+f1, -y(1)+f2]';
end
```

We then use the following code, modified from earlier to produce a plot of our orbits in the phase plane.

Note that we are now also calculating the mean orbits (the statistical mean, which coincides with the collection of means at each discrete nodal point t_k, after the executed number of runs). In fact during the different runs, due to the strategy of time discretization applied automatically by '*ODE45*', the discrete nodal points change, and thus, we approximate the local mean by the nearest values to some t of a fixed discretization, i.e. sum for the discrete points $t_k \approx t$. We use values stored from each iteration of the '*ODE45*' algorithm.

```
format long e
```

```
options = odeset('RelTol', 1e-5,'AbsTol', [1e-5 1e-5]);
title('Mean orbit')
ylabel('y')
xlabel('x')
hold on
sum1=0;
sum2=0;
sumA=zeros(100);
sumB=zeros(100);
for k = 1 : 200
    [t, u] = ode45('ex5d', [0, 10], [0.1, 0.1], options);

s=size(u(:,1));
ll=size(u);
l=ll(1);
%the number of time nodes not the same
%we keep only tfinal=10 which coresponds at the last line ...
   of u
%aa(k)=u(l,1);
for j=1:l
u1sumA(j)=sumA(j)+u(j,1);
u2sumB(j)=sumB(j)+u(j,2);
end
end
sum11=0;
sum22=0;
for jj=1:l
    ufin(jj,1)=sum11+u1sumA(jj);
    ufin(jj,2)=sum22+u2sumB(jj);
end
plot(ufin(:,1),ufin(:,2)) %Plot mean orbits
```

First we plot all the orbits for each run of the algorithm ($k = 200$). Doing this we obtain the following plot. Note here that we have inserted small random perturbations in the ODE system, while the initial conditions remain deterministic (i.e. independent from each realization of experiment).

It can now be seen in Figure 4.8 that we no longer have perfect orbits, which is due to the small perturbations in the ODE system where we inserted our randomness, so we now take the mean orbit as our final solution. Doing this we get the following plot shown in Figure 4.9 which is a similar orbit to those seen earlier where we didn't have any randomness.

Now that we have investigated the result of randomness in the ODE system, we now want to insert randomness in the initial conditions and investigate the effect this has on the orbits. To do this we simply replace our initial conditions in the above code with randomly selected values from N(0,1) f_1 and f_2.

We now have initial conditions of the form

$$1 + (10)^{-3} \times normrnd(0, 1).$$

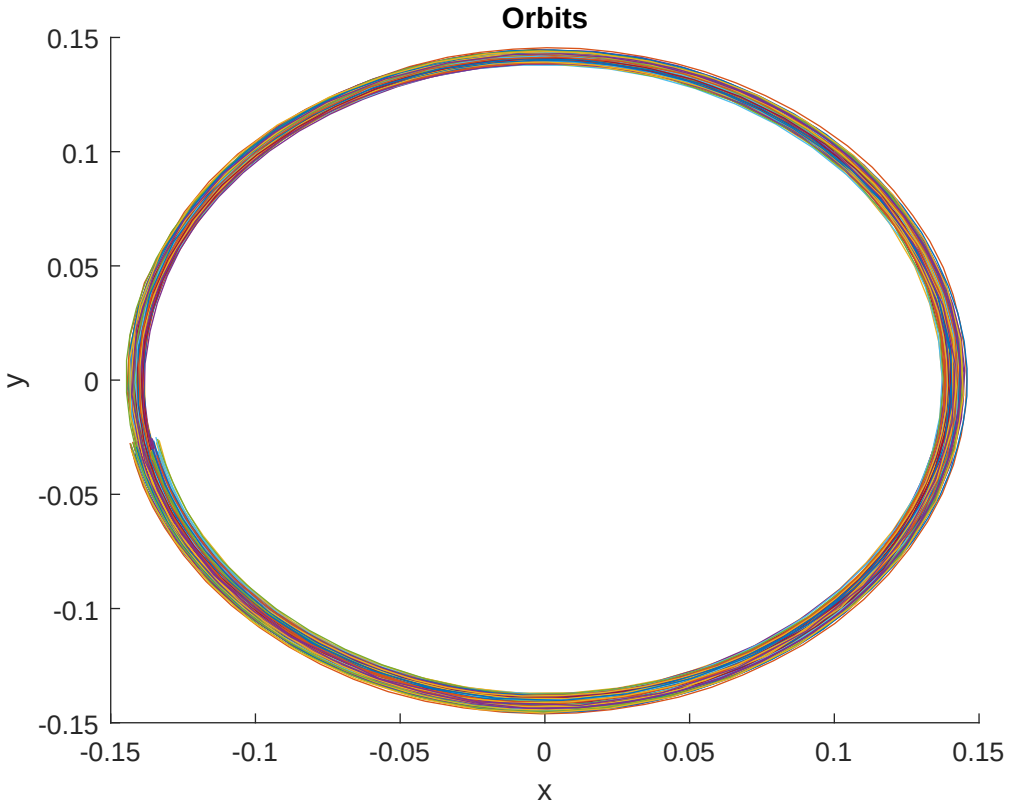


Figure 4.8: All orbits in phase plane for harmonic pendulum with randomness in the ODE system.

Note that f_1 and f_2 are different since we choose a different random value (randomly chosen value and thus varying) for each initial condition at each iteration.

This produces the following mean orbit, as expected this is an ellipse centred at $(0, 0)$.

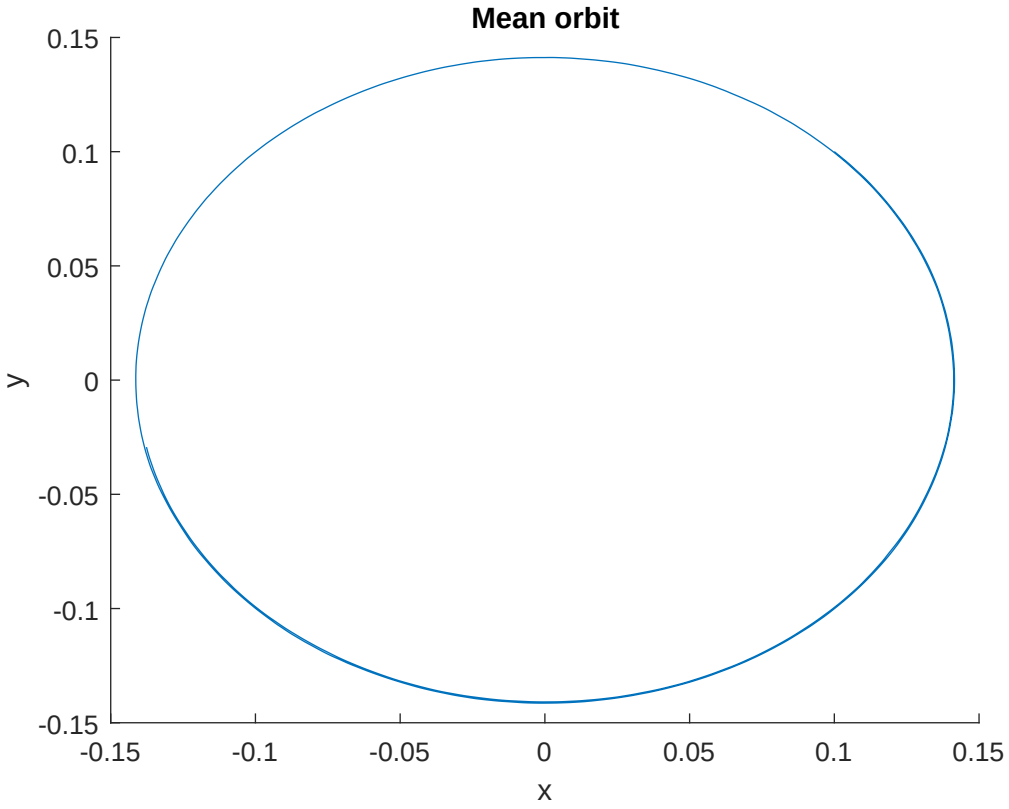


Figure 4.9: Mean orbits in the phase plane for harmonic pendulum with randomness in the ODE system.

Now let us change some of the parameters to see what effect this has on the system and how the randomness effects our results under different values of parameters. Let us start with larger perturbations. i.e instead of using $10^{-3} \times B(1)$ let us remove the 10^{-3} and replace it with 10^{-1} so that we have essentially made the perturbations much much larger. We can use the same MATLAB code given earlier but we edit the values of the parameters accordingly. We can see all of the orbits plotted in the (x, y) phase plane below. Note that we haven't changed the initial conditions here, these remain at 0.1, but we have changed the value of the perturbations in the function file. So we have the following system

$$\dot{x} = y + 10^{-1} \times B(1)$$
$$\dot{y} = -x + 10^{-1} \times B(1).$$

It is also important to note here that for these experiments we will reduce the tolerance in the '*ODE45*' solver. This is because when we increase the size of the perturbations, the solver will take much longer to solve, so for computational purposes we will reduce the tolerance from 10^{-5} to 10^{-3}. This won't effect our results much at all, but will improve computational time a lot.

It can be seen from Figure 4.11 that when we increase the randomness at the ODE side of the system, the results are much more chaotic, and the elliptical shape that we seen earlier has become much more distorted. This can be seen

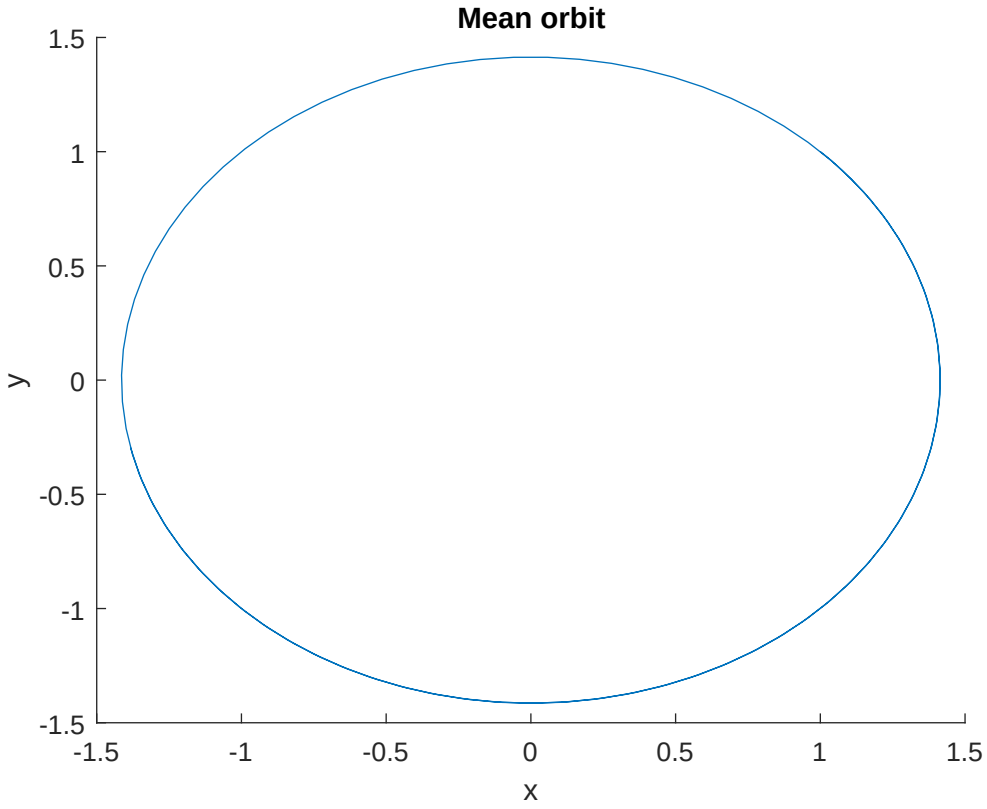


Figure 4.10: Mean orbits in the phase plane for harmonic pendulum with randomness in the initial conditions.

more clearly when we study the graph of the mean orbit in this example which can be seen below in Figure 4.12. We can see now that the mean orbit is not even close to being a an ellipse. This is because the randomness in the system is far too large and dominates the ODE, meaning we get the chaotic looking solutions that can be seen in Figures 4.11 and 4.12.

If we compare the mean orbit in Figure 4.12 to that in Figure 4.9 where we did the same experiment but with smaller random values we can see that by increasing the size of the perturbations in the ODE system we receive somewhat chaotic results, it is clear that the randomness has dominated the system and the results the reflect this. Next let us try the same experiment, but this time increasing the size of the perturbations in the initial conditions. Here we shouldn't expect to see as much difference to earlier experiment since the ODE is still the same and we are only changing the size of the initial conditions. The phase plane showing all of the orbits for this experiment can be seen in Figure 4.13 which shows that when we increase the value of the initial conditions we still get our elliptical shapes, but we get ellipses with much larger radii. Note that here we can keep the high tolerances because the solver will not be effected by the values of the initial conditions and our computational time will not be effected as it was in the earlier experiment.

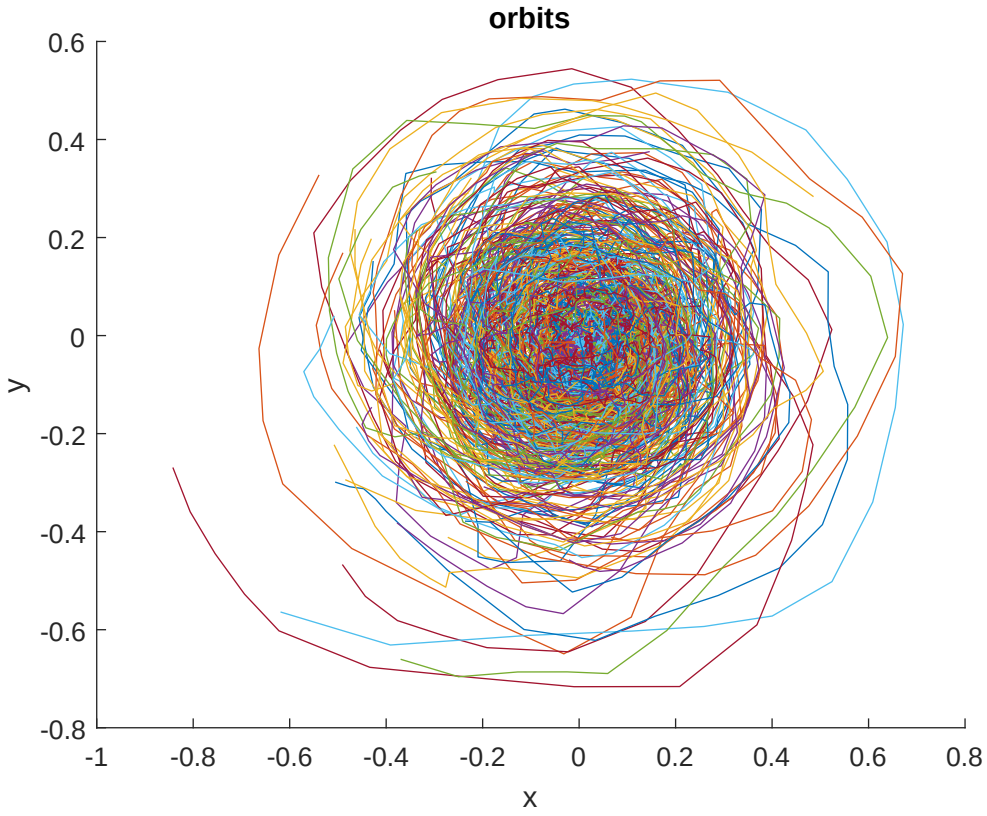


Figure 4.11: All orbits in the phase plane for harmonic pendulum with larger randomness in the ODE system.

The mean orbits for this experiment can be seen below in Figure 4.14. It is almost identical to the previous experiment we did with random initial conditions, however the radius of the ellipse here is much much larger. Note that we used initial conditions of $0.1 + 10 * normrnd(0, 1)$ to amplify the effect that this change had on our graphs.

Randomness in ω

Another interesting result occurs when we insert randomness at the ω term. For the purposes of the following runs let us choose $\omega = 0.5 + 10^{-1} \times B(1)$ where $B(t)$ represents the Brownian motion or randomness. Note that we run all these experiments with tolerance $10^{-}3$, this is because the tolerance doesn't really effect these runs and the code runs quickly enough at this tolerance level. Figure 4.15 shows that we get ellipses with varying radius due to the randomness in ω. The mean orbit can then be seen in figure 4.16.

Next we want to investigate what happens when we increase the size of the randomness in the ω term, so we run the next experiment with $\omega = 0.5 + B(1)$. The result of this can be seen in Figure 4.17. We notice that the results become even more chaotic compared to those in the previous experiment (Figure 4.15). The mean orbit for this experiment can then be seen in Figure 4.18.

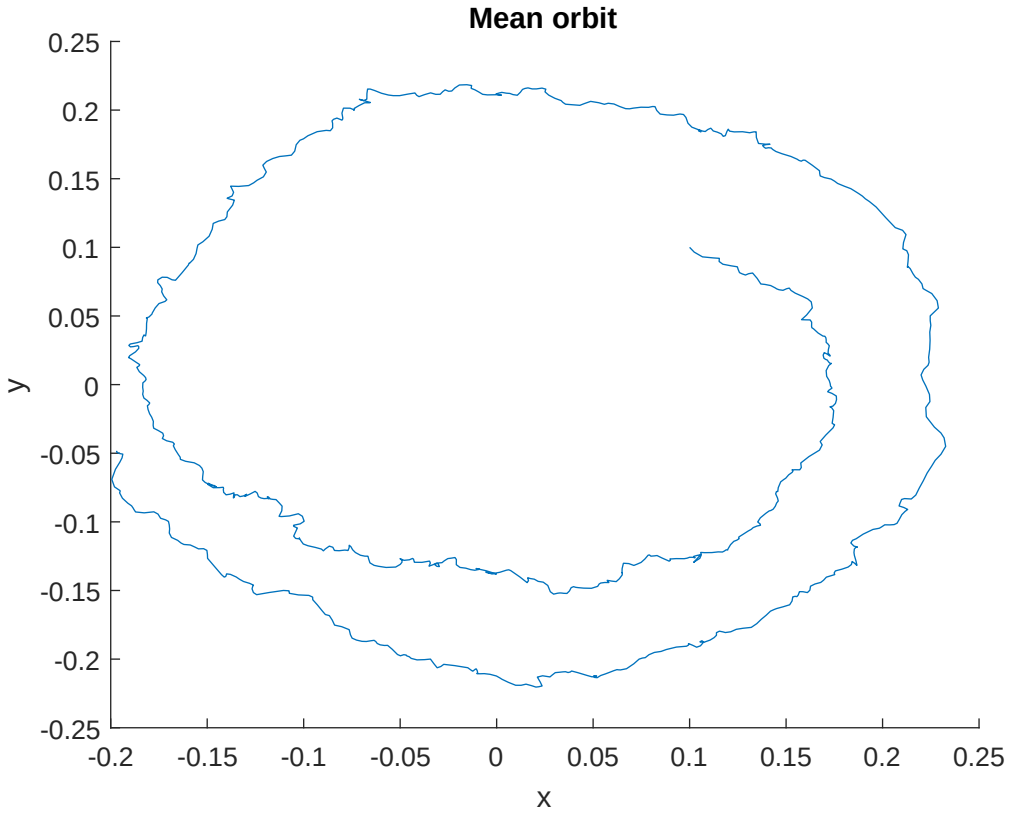


Figure 4.12: Mean orbit in the phase plane for harmonic pendulum with larger randomness in the ODE.

We observe that in this set of experiments, we inserted randomness to the coefficients of the ODE, i.e. to the system's matrix, and this can change even the stability profile of the ODE; even a very a small change on a matrix changes the eigenvalues of it, this change being critical when the equlibrium point stability character changes.

Solutions with respect to time t

We also want to see how the solution behaves with respect to the time t. First let us plot the solutions (t, y) and (t, x) when there is no randomness, we will then insert some randomness to see how this effects the solutions to the system with respect to the time t.

The plots of (t, y) and (t, x) can be seen in Figure 4.19.

Now we also want to see how these oscillatory solutions are effected by the presence of randomness. Let us see first the effect of randomness in the initial conditions in Figure 4.20. Here we used randomness of $10^{-3} \times B(1)$ and we can see that the randomness changes the points at which each oscillation changes direction. This can be seen in Figure 4.20. Note that we will only present the solution of (t, y) since the effect is identical on the (t, x) solution.

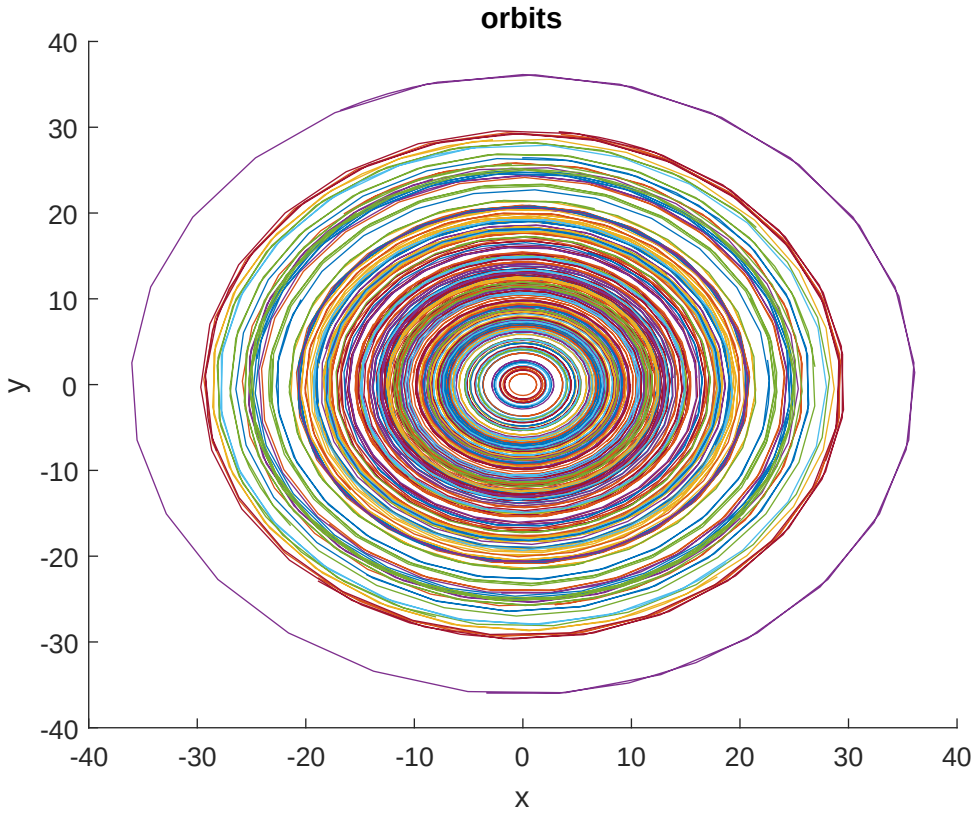


Figure 4.13: All orbits in the phase plane for harmonic pendulum with larger randomness in the initial conditions.

Next we investigate the effect that placing randomness in the system has. Again we use randomness of 10^{-3} like we did in the earlier experiments. It can be seen in Figure 4.21 that the randomness doesn't have much effect on the oscillatory behaviour of the solution. Again we only present the (t, y) solution here.

What have we learnt from these experiments

Throughout this section we have inserted varying degrees of randomness at different points in the system. We noted that by inserting randomness in the initial conditions we did not observe much change in the orbits at all, however when we inserted randomness at the ODE and in the ω term we found the size of this randomness effected greatly the resulting phase plane orbits.

When we used randomness of the order 10^{-3} we observed that the system was able to cope and there was little or no change in our final results. As we increased the size of this randomness up to the order of 10^{-1} and above we noticed that it started to dominate the system and our results became much more chaotic.

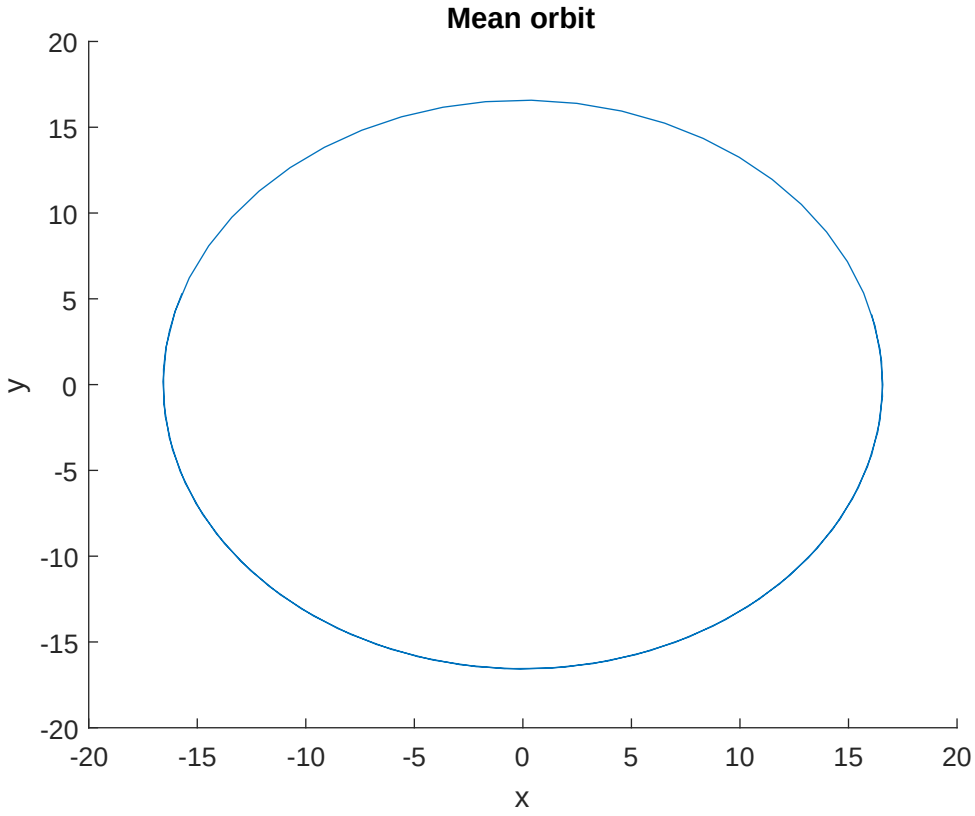


Figure 4.14: Mean orbit in the phase plane for harmonic pendulum with larger randomness in the initial conditions.

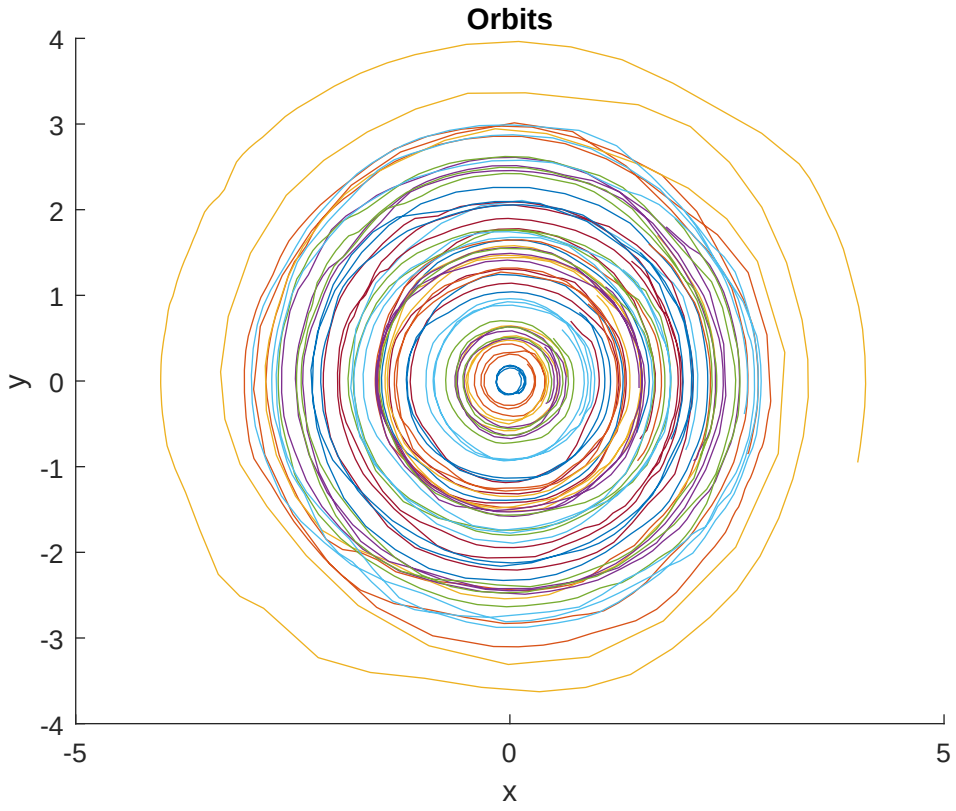


Figure 4.15: All orbits with randomness $10^{-1} \times B(1)$ in ω.

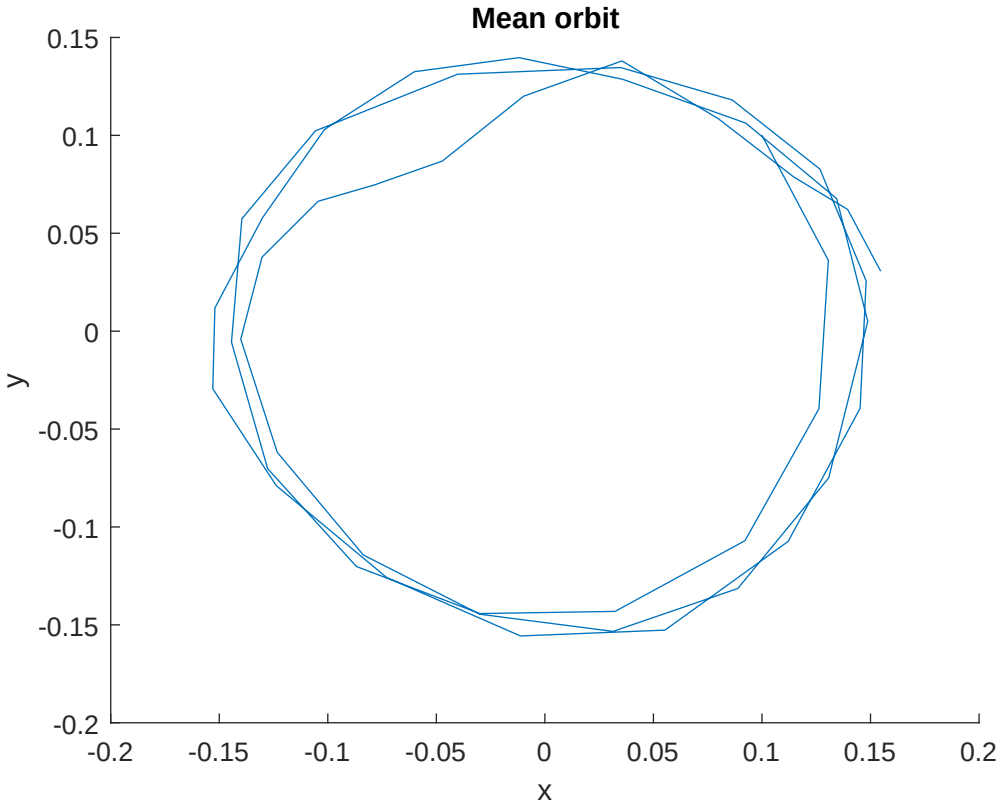


Figure 4.16: Mean orbit with randomness $10^{-1} \times B(1)$ in ω.

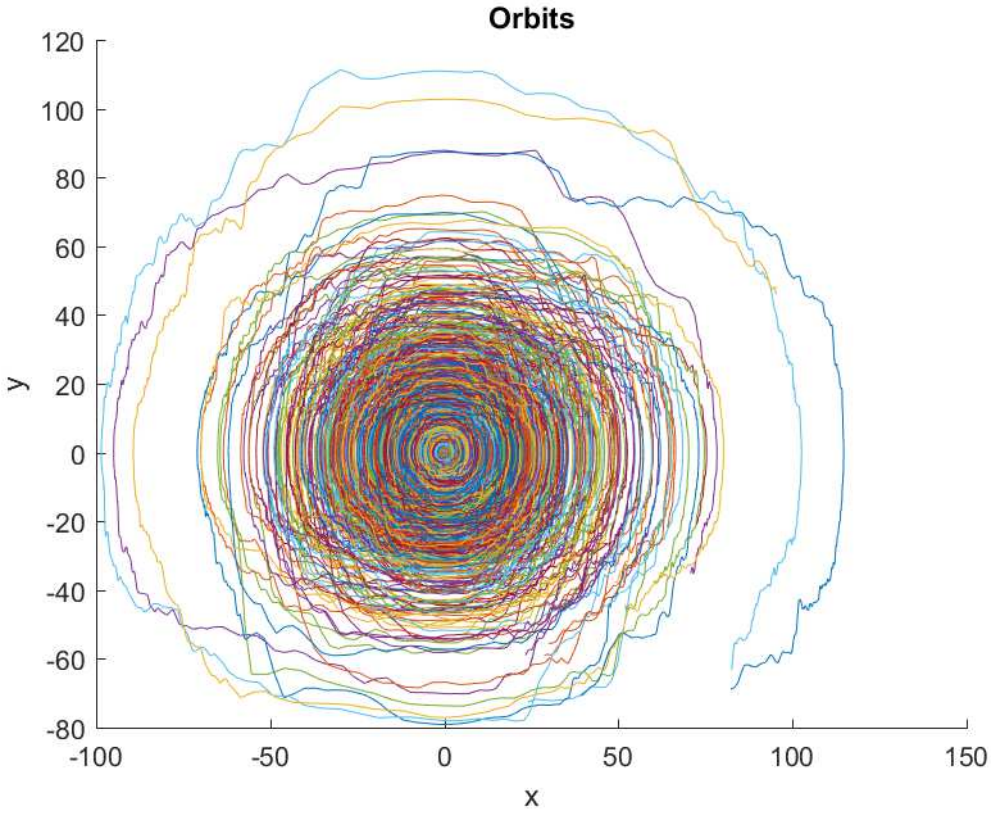


Figure 4.17: All orbits with randomness $B(1)$ in ω.

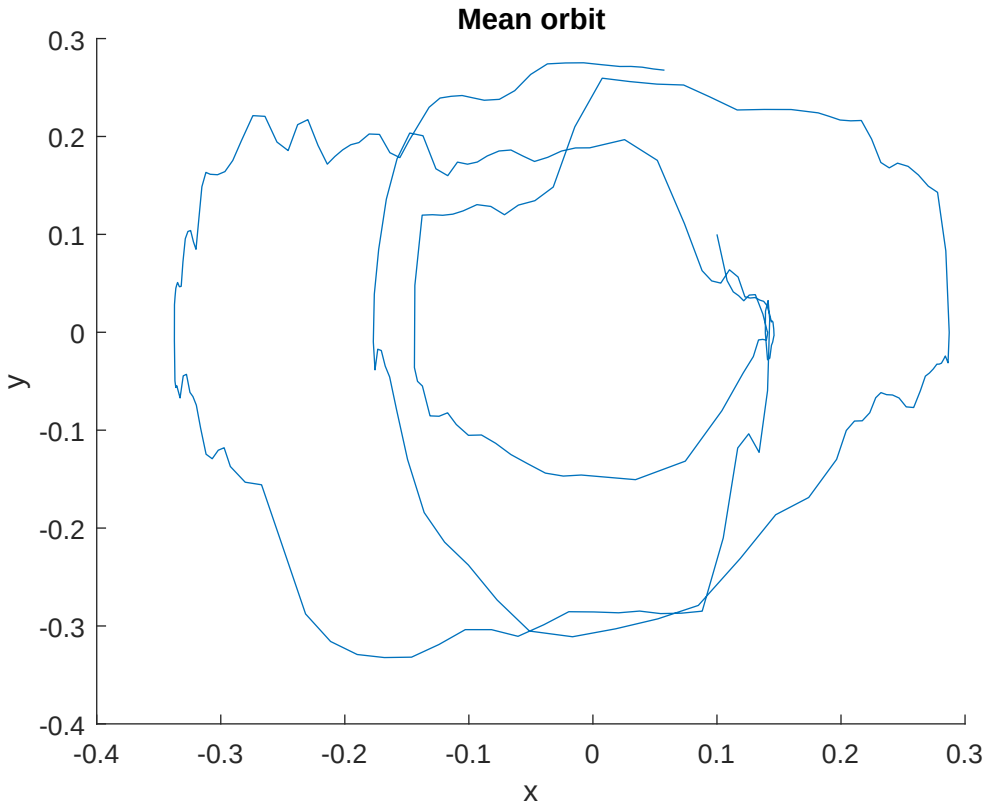
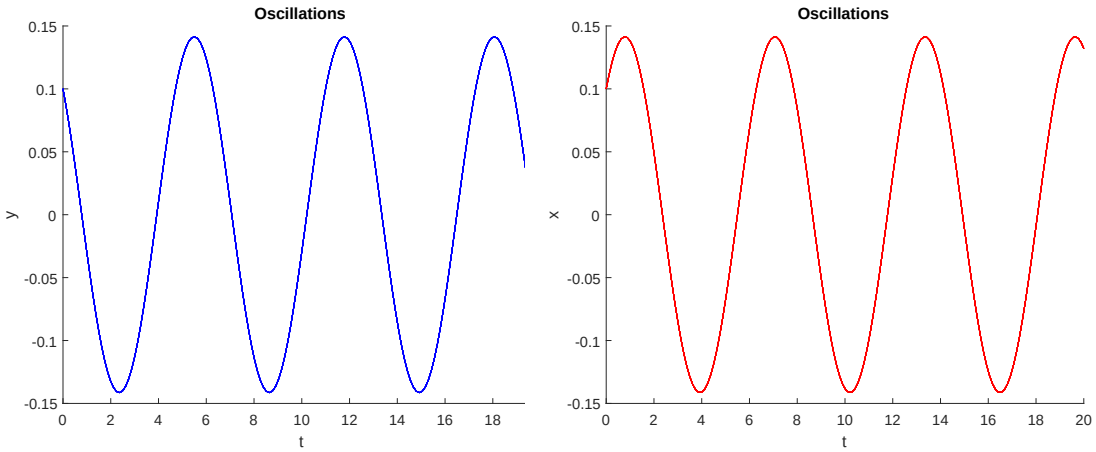


Figure 4.18: Mean orbits with randomness $B(1)$ in ω.



(a) Solution with respect to t, (t, y). (b) Solution with respect to t, (t, x).

Figure 4.19: Solutions with respect to t over the interval $[0, 20]$.

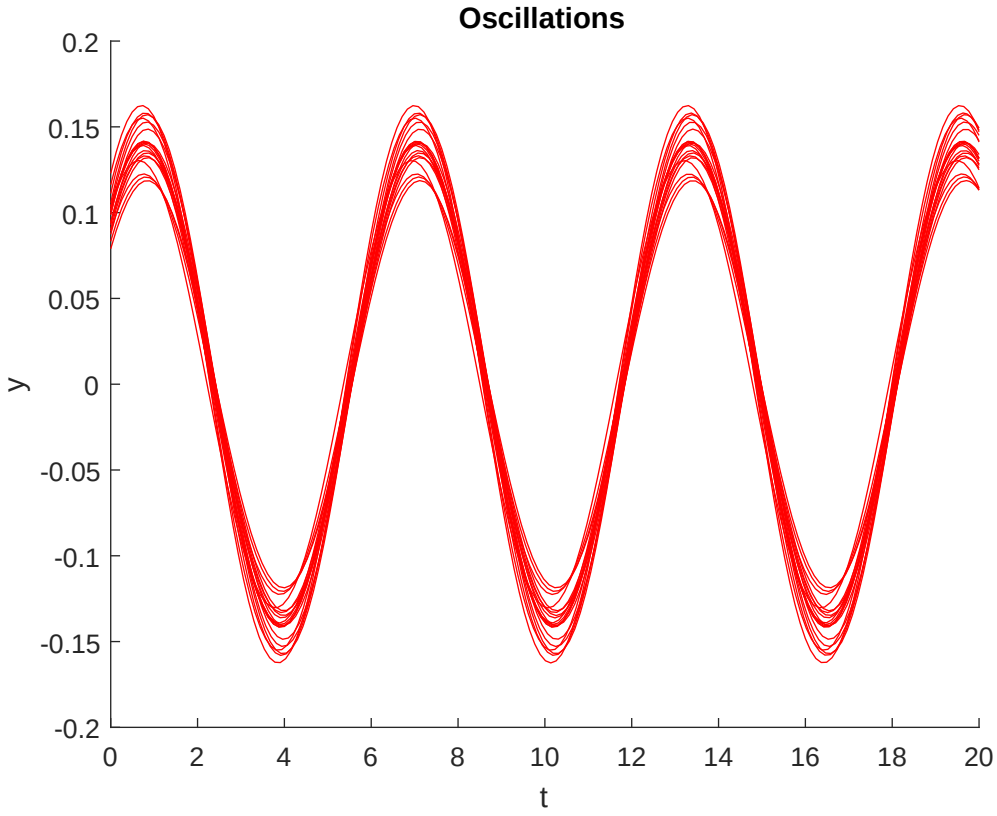


Figure 4.20: Solution (t, y) with randomness inserted at the initial conditions.

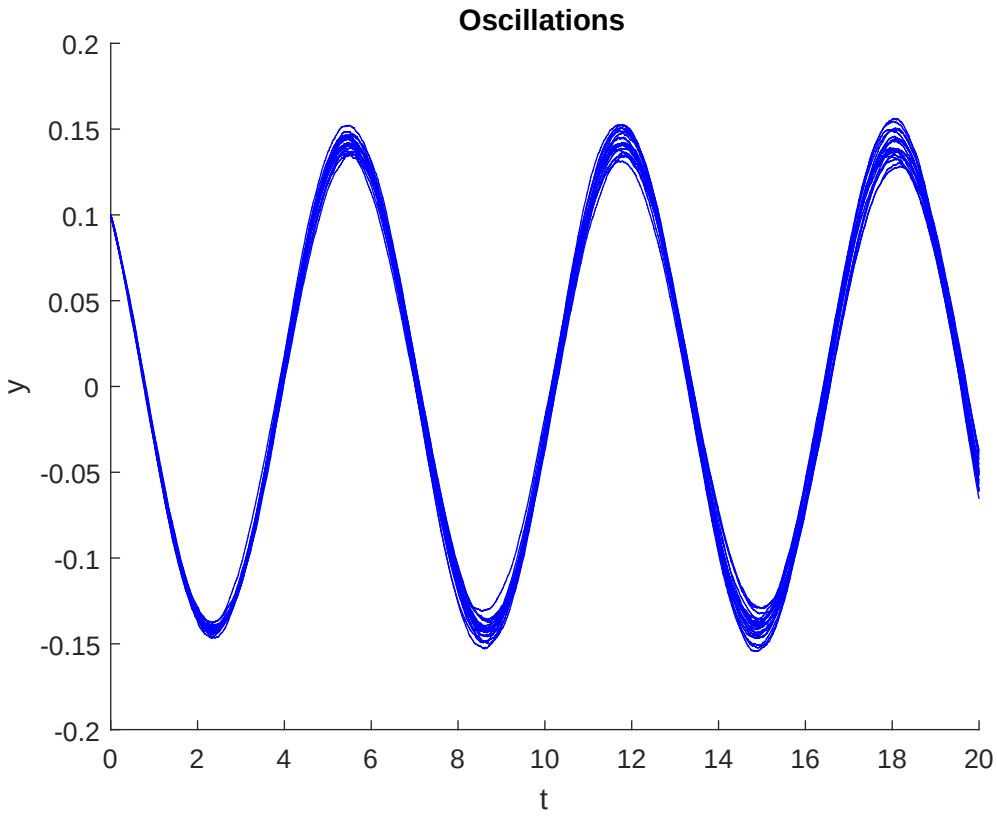


Figure 4.21: Solution (t, y) with randomness inserted at the ODE system.

Conclusion

In Chapter one we presented various related iterative methods related to approximating the solutions to large sparse linear and non-linear systems when the problem is equivalently posed as an optimization problem. We proposed the basic version of these algorithms, as this was a main aim of this Thesis. The algorithms we presented have many modified versions which can be found in [11].

One of these is a restarted version, in the restarted version of these iterative methods, where we basically restart our procedure after k iterations. We do this because for larger systems and more difficult ones it is more beneficial in terms of computational cost to discard all of the previous iterations, as storing these is computationally expensive. In most cases this does not effect the convergence of such methods.

In Chapter two, we presented many examples of solving optimization problems in MATLAB [8], [9]. We provided detailed explanations of how to solve a variety of optimization problems using the inbuilt toolbox. In further work it would be useful to explore solving problems in higher dimensions, particularly non-linear problems. It would also be useful to explore more the algorithms and solvers that are contained in MATLAB's optimization toolbox.

In Chapter three, we presented the Travelling Salesman Problem [7], a classical mathematical problem where we seek to find an optimal route around N cities. We aimed to present the TSP as an optimization problem which we did and also provided two examples to show how the problem becomes much much more difficult as the value of N becomes larger. In further work it would be useful to explore the methods of solution [2],[3] in more detail and try to implement such methods computationally in MATLAB.

Chapter four was based on Monte Carlo methods [14], we provided a simple example to demonstrate how to apply a Monte Carlo procedure and then went on to apply a Monte Carlo method to a system based on a harmonic pendulum. We changed various parameters including the tolerance in the solver, the size of the initial conditions and the size of the randomness in our system. We investigated the effect of each of these and the results relating to the randomness in the system were particularly interesting.

In further work we would aim to provide more examples of applying randomness to a variety of ODE systems with varying dimensions.

Bibliography

[1] Mordecai Avriel, *Nonlinear programming: Analysis and methods*, Courier Corporation, 2003.

[2] Marco Dorigo, *Ant colonies for the travelling salesman problem*, IEEE Transactions on Evolutionary Computation **1** (1997).

[3] Greco F. (ed.), *Travelling salesman problem*, I-Tech, 2008.

[4] Gene H. Golub and Charles F. Van Loan, *Matrix computations*, 3rd ed, Johns Hopkins studies in the mathematical sciences, Johns Hopkins University Press, 1996.

[5] Magnus R. Hestenes and Eduard Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards **49** (1952).

[6] Roger A. Horn and Charles R. Johnson, *Matrix analysis*, 2nd ed., Cambridge University Press, 2013.

[7] Bernhard Korte and Jens Vygen, *The travelling salesman problem*, Combinatorial Optimization. Algorithms and Combinatorics **21** (2008).

[8] Roy H Kwon, *Introduction to linear optimization and extensions with matlab*, Operations research series, CRC Press, 2014.

[9] Achille Messac, *Optimization in practice with matlab for engineering students and professionals*, Cambridge University Press, 2015.

[10] Bernt Oksendal, *Stochastic differential equations: an introduction with applications*, 6th ed, Universitext, Springer, 2010.

[11] Yousef Saad, *Iterative methods for sparse linear systems*, 2nd ed., Society for Industrial and Applied Mathematics, 2003.

[12] J. Snyman, *An introduction to basic optimization theory and classical and new gradient-based algorithms*, Practical Mathematical Optimization, Springer, 2005.

[13] J. Stoer and R. Bulirsch, *Introduction to numerical analysis*, 3rd ed, Texts in Applied Mathematics 12, Springer New York, 1993.

[14] Dirk. P. Kroese, Tim Brereton, Thomas Taimre, and Zdravko I. Botev, *Why the monte carlo method is so important today*, Wiley Interdisciplinary Reviews: Computational Statistics **6** (2014).